(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
3 January 2002 (03.01.2002)

PCT

(10) International Publication Number
WO 02/01833 A1

(51) International Patent Classification[7]: H04L 29/06

(21) International Application Number: PCT/US00/17844

(22) International Filing Date: 28 June 2000 (28.06.2000)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant *(for all designated States except US)*: MICROSOFT CORPORATION [US/US]; One Microsoft Way, Building 114, Redmond, WA 98052 (US).

(72) Inventor; and

(75) Inventor/Applicant *(for US only)*: ZINTEL, William, Michael [US/US]; 7122 N.E. 188th Court, Kenmore, WA 98028 (US).

(74) Agent: WIGHT, Stephen, A.; Klarquist, Sparkman, Campbell, Leigh & Whinston, LLP, One World Trade Center, Suite 1600, 121 SW Salmon Street, Portland, OR 97204 (US).

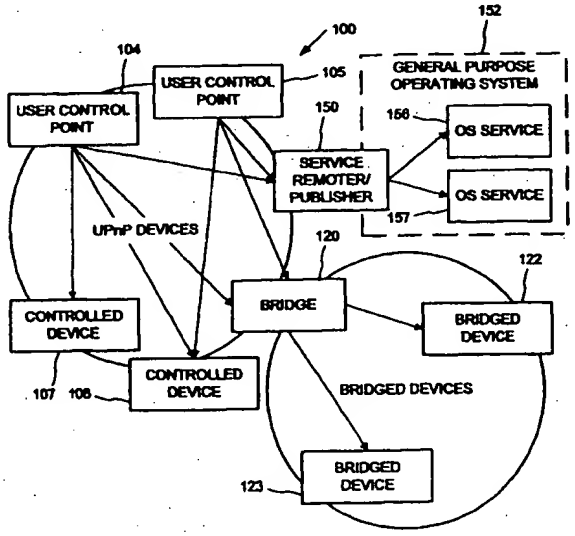
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: REMOTING GENERAL PURPOSE OPERATING SYSTEM SERVICES VIA A PEER NETWORKING DEVICE CONTROL PROTOCOL



(57) Abstract: An operating system (OS) services-to-peer networking connectivity adapter exposes services of a general purpose operating system to remote control via a peer networking device control protocol, which allows reuse of the operating system service without reprogramming the service as a "controlled device" per the peer networking device control protocol. The adapter includes a publishing service that implements components of a controlled device and operates to emulate a set of controlled devices that expose the functionality of the OS services. A service remoter converts between a publishing application programming interface (API) of the publishing service and the established operating system API of the OS services.

WO 02/01833 A1

REMOTING GENERAL PURPOSE OPERATING SYSTEM SERVICES VIA A PEER NETWORKING DEVICE CONTROL PROTOCOL

TECHNICAL FIELD

This invention relates generally to remote access to and control of general purpose operating system services, and more particularly to remoting such services via a peer networking device control protocol.

BACKGROUND AND SUMMARY

Present day, general purpose operating systems provide numerous services for use by application programs. An operating system is software that generally controls the allocation and usage of hardware resources on a computer, such as memory, central processing unit (CPU) time, disk space, and peripheral devices. General purpose operating systems provide an operating platform or environment on which application software for a wide variety of purposes can be run. Examples of popular general purpose operating systems include Microsoft Windows operating systems (e.g., Windows 95, 98, NT, 2000); the Mac OS of Apple Computer, Inc.; UNIX or Linux available from various vendors; and BeOS from Be, Inc.

A service has generally been defined to be a program, routine, or process that performs a specific system function to support other programs, particularly at a low (close to the hardware) level. Typically although not necessarily, the services do not provide a user interface (UI) or interact directly with the user (e.g., display messages in a window or dialog box, or receive input from keyboard or mouse) during normal processing (except error messages and like interaction when exceptions occur, and administration or configuration of the service), but instead

typically operate under programmatic control of an application program with which the user directly interacts. Examples of services provided in the Microsoft Windows® operating systems include date and time clock, file and printer sharing, print spooling, electronic mail, fax, telephony, file synchronization (e.g., with mobile devices), file backup and archiving, audio and video file streaming (e.g., to a peripheral play-back device, such as speakers or a monitor), audio and video codecs, file compression/decompression, problem diagnosis and support, event logging, user and software configuration administration, security and access control, remote access (RAS), networking (Internet socket, FTP, HTTP, TCP/IP, NetBIOS, etc.), name and address (DHCP, DNS), remote procedure call (RPC) and named pipe, dynamic peripheral configuration (Plug and Play), and application scheduling (e.g., Windows 95 system agent), among others. In addition to the services provided with an operating system by its vendor, some operating systems (e.g., Microsoft Windows NT) also support an extendible services architecture that permits plugging in or installing later-developed services (e.g., Windows NT Services, Unix Daemon).

Operating system services typically are exposed for programmatic control and use by locally executing application programs through an application programming interface (API) of the operating system. The application program typically provides a user interface with which the user interacts to effect application-specific task work. The application program may utilize the operating system services, and programmatically controls the services through the APIs. For example, a "music jukebox" application program could provide a user interface having a "graphics equalizer" and "playlist" display, and start, stop and pause controls with which the user

interacts to play music audio files. The application program, in turn, utilizes audio streaming services of the operating system, which the application program controls via an API. The operating system API typically operates using a local procedure call, software interrupt, or like mechanism.

Present industry trends (e.g., rapidly decreasing costs of computing and networking technologies) are leading towards embedding computing and networking capabilities into the design of many specific-purpose devices in the home, office and public places. Examples include digital cameras; audio/video receivers, recorders and players; printers; home appliances; lighting systems; heating, ventilation and air-conditioning (HVAC) equipment; security systems; telephony equipment; and etc. The combination of inexpensive and reliable shared networking media with a new class of small computing devices has created an opportunity for new functionality based mainly on the connectivity among these devices. This connectivity can be used to remotely control devices, to move digital data in the form of audio, video and still images between devices, to share information among devices and with the Internet and to exchange structured and secure digital data to support things like electronic commerce. The connectivity also enables many new applications for computing devices, such as proximity-based usage scenarios where devices interact based at least in part on geographical or other notions of proximity. These developments are occurring at the same time as more people are becoming connected to the Internet and as connectivity solutions are falling in price and increasing in speed. These trends appear to lead towards a world of ubiquitous and pervasive networked computing,

where computing and networking are built into all types of devices in the user's home, work and public environments.

With pervasive networked computing, the general use scenario changes from one where the user interacts directly with a device that performs some work (e.g., by pressing buttons on a control panel), to one where the user interacts with one device that provides user interactivity and remotely controls another device doing the work. For example, the user directly interacts with a universal remote controller, cell phone or handheld/palm top/tablet computer to operationally control other devices in their immediate environment, such as a vending machine, video monitor, printer, door opener, or etc.

Recently, several peer networking device control protocols have been introduced to support this new remote control use scenario expected to become typical of pervasive computing. These include the Home Audio Video Interoperability (HAVi) protocol developed by the Havi Organization (formed by Grundig AG, Hitachi, Ltd., Matsushita Electric Industrial Co., Ltd. (Panasonic), Royal Philips Electronics, Sharp Corp., Sony Corp., Thomson Multimedia and Toshiba Corp); the JINI protocol developed by Sun Microsystems, Inc.; and the Universal Plug and Play (UPnP) protocol of the UPnP Forum and Microsoft Corporation. In general, these peer networking device control protocols are designed to allow devices to expose their operational functionality to remote control from devices directly operated by a user.

As discussed above, general purpose operating systems provide a rich set of services to application programs running on the operating system via locally accessible APIs. In pervasive computing, peer networking device control protocols allow remote control of operational

functionality of typically specific purpose devices. It would generally be advantageous to also remotely control the general purpose operating system services in pervasive computing environments. However, because the services are programmed to be exposed through the operating system APIs and not via peer networking device control protocols, the services are not available to user control devices through these protocols.

In the UPnP protocol, legacy peripheral devices (e.g., scanner, printers, data storage drives, etc.) connected to a host computer can be exposed for remote control from a "user control point" device via a "UPnP bridge," which acts as a converter or adapter between the UPnP peer networking connectivity protocol and a host-peripheral connectivity protocol. The UPnP bridge generally runs on the host computer acting as a controlled device in place of a "bridged peripheral device" to respond to addressing, discovery, description, and control messaging in the UPnP protocol from user control point devices, and in turn interacts accordingly with the bridged peripheral device using the bridged peripheral's host-peripheral connectivity protocol. Again however, because general purpose operating system services are programmed to be exposed through operating system APIs, the services are not available to remote control via a UPnP bridge to a host-peripheral connectivity protocol.

The present invention provides a way to expose general purpose operating system services to remote control via a peer networking device control protocol, which allows reuse of the operating system service without reprogramming the service as a "controlled device" per the peer networking device control protocol. In accordance with the invention, an adapter runs on the general purpose operating system and converts

between the peer networking device control protocol and operating system APIs, so as to remote the general purpose operating system's services.

In an implementation of the invention described herein, the adapter includes a publishing service and a service remoter. The service remoter operates to remote operational functionality of general purpose operating system services through the publishing service. The publishing service supports a publishing API via which application programs running on the operating system as well as the service remoter can interact with the publishing service. The publishing service operates to expose operational functionality of the application programs and the services that interact with the publishing service through the publishing API as "controlled devices" under the peer networking device control protocol. The publishing service acts as the "controlled devices" per the peer networking device control protocol by appropriately responding on behalf of the remoted applications and services to addressing, discovery, description and control requests made by user control point devices in the peer networking device control protocol. The publishing service further converts these peer networking device control protocol requests to cause appropriate controlled operation of the services through the service remoter, which in turn interacts directly with the services through their APIs to effect the controlled operation.

Additional features and advantages will be made apparent from the following detailed description of the illustrated embodiment which proceeds with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figures 1 and 2 are block diagrams of a device architecture per Universal Plug and Play using user control points, controlled devices,

bridges, and service remoter/publisher for connectivity between devices and services.

Figure 3 is a block diagram of a general purpose operating system architecture with service remoter/publisher for remoting operating system services per Universal Plug and Play in accordance with an implementation of the invention.

Figure 4 is a block diagram of an implementation of the service remoter/publisher of Figure 3 in the Microsoft Windows operating system architecture.

Figure 5 is a block diagram of a device model per Universal Plug and Play.

Figure 6 is a block diagram illustrating example devices conforming to the device model of Figure 5.

Figure 7 is a block diagram illustrating device state synchronization using a state table and eventing.

Figure 8 is a block diagram illustrating device addressing.

Figure 9 is a block diagram of a programmatic interface-to-network messaging adapter or Rehydrator in the device control model of Figure 5.

Figure 10 is a general data flow diagram of the Rehydrator of Figure 9 in the device control model of Figure 5.

Figure 11 is a block diagram of an implementation design of the Rehydrator of Figure 9.

Figures 12 and 13 are block diagrams illustrating an internal software architecture of the user control point and controlled device in the device control model of Figure 5.

Figure 14 is a block diagram illustrating an internal software architecture of a combined bridge and user control point in the device control model of Figure 5.

Figure 15 is a data flow diagram illustrating a typical browsing protocol sequence in the device control model of Figure 5.

Figure 16 is a listing showing a layout of a description document in the device control model of Figure 5.

Figure 17 is a listing of an exemplary icon list of a Description Document in the device control model of Figure 5.

Figure 18 is a listing of an exemplary service control protocol declaration in a Description Document in the device control model of Figure 5.

Figures 19 and 20 are a listing of an XML schema for a Service Control Protocol Declaration Language used in the device control model of Figure 5.

Figure 21 is a block diagram of an eventing model used in the device control model of Figure 5.

Figure 22 is a data flow diagram illustrating subscription, notification and unsubscription in the eventing model of Figure 21.

Figure 23 is a block diagram of a computer system that may be used in the device control model of Figure 5.

Figure 24 is a block diagram of a device having embedded computing and networking capability per universal-plug-and-play (UPnP) standards that may be used in combination with the computer system of Figure 23 in the device control model of Figure 5.

Figure 25 is a block diagram of a software architecture per UPnP standards in the embedded computing device of Figure 24

Figure 26 is a data flow diagram of a process for automatic network introduction of the embedded computing device of Figure 24 into an ad hoc computer network environment per the UPnP protocol.

Figure 27 is a data flow diagram of a process for automatic network introduction of the embedded computing device of Figure 24 into a configured computer network environment per the UPnP protocol.

Figure 28 is a block diagram of a software architecture of a client device per UPnP standards having embedded computing and networking capability that may be used in the device control model of Figure 5.

Figure 29 is a block diagram of an exemplary home or office pervasive computing environment having a variety of computers as per Figure 23 and embedded computing devices as per Figure 24 interconnected per UPnP standards that may be used in the device control model of Figure 5.

Figures 30 through 40 are program listings of interfaces used in the Rehydrator implementation design of Figure 11.

Figure 41 is a flow diagram illustrating automatically installing and configuring a UPnP bridge upon attaching or connecting a peripheral on a host personal computer.

DETAILED DESCRIPTION

The following detailed description is directed toward adapting general purpose operating system (OS) services for operation within a distributed device control model having peer networking connectivity. In one described implementation, this OS service-to-peer networking control protocol adapter is used in a device architecture 100 (Figure 1),

connectivity model, and device control protocol proposed by Microsoft Corporation, called Universal Plug and Play ("UPnP").

Universal Plug and Play

Universal Plug and Play (UPnP) is an open network architecture that is designed to enable simple, ad hoc communication among distributed devices and services from many vendors. UPnP leverages Internet technology and can be thought of as an extension of the Web model of mobile Web browsers talking to fixed Web servers to the world of peer-to-peer connectivity among mobile and fixed devices. UPnP embraces the zero configuration mantra of Plug and Play (PnP) but is not a simple extension of the PnP host/peripheral model.

The cost, size and battery consumption of computing technology--including processing, storage and displays--continues to fall. This trend is enabling the evolution of stand-alone, single or limited function computing devices such as digital cameras, audio playback devices, smart mobile phones and handheld computers. Concurrent with this, the economical storage and transmission of digital audio, video and still images is enabling highly flexible models for managing entertainment content.

While many of these devices are capable of useful stand-alone operation, seamless connectivity with the PC can enhance the value to the customer of both stand-alone devices and the PC. Good examples of this synergy are digital image capture combined with PC image manipulation, storage and email transfer/Web publishing and information synchronization between a PC and a handheld computer or smart mobile phone.

Since many of these devices, and the PC itself, are mobile, a suitable communication architecture must enable a highly dynamic connectivity

model and must enable peer-to-peer interoperability among arbitrary combinations of devices.

The Internet has created a widespread awareness of the value of simple, universal communication that is independent of the underlying transmission technology and independent of technology from any single vendor.

UPnP makes it possible to initiate and control the transfer of bulk data (e.g. files) or A/V data streams from any device on the network, to any device on the network, under the control of any device on the network. UPnP enables the ad hoc addition or removal of devices on the network, and it enables multiple controlling devices to remain in sync with each other.

UPnP reuses existing protocols and technology whenever possible. The transition to this highly connected (and connectable) world will not occur overnight. UPnP builds on existing Internet protocols, but accommodates devices that cannot run the complete UPnP protocol suite. UPnP provides an architecture that enables legacy devices to communicate with UPnP devices.

IP internetworking has been chosen as a UPnP baseline due to its proven ability to span different physical media, to enable real world multiple vendor interoperation and to achieve synergy with the Internet and home and office intranets. Internet synergy enables applications such as IP telephony, multiple player games, remote control of home automation and security, Internet based electronic commerce, in addition to simple email and Web browsing. UPnP's scope includes remote control of devices and bulk data transfer, and can be easily extended to specify A/V streaming.

UPnP's media independence enables a great deal of flexibility in the packaging of products. UPnP enables an A/V system to be controlled through an A/C power communications technology, while the transmission of A/V streams among the components is analog or digital. One of the controllers of this system could be on the television, while another is on a PC, and yet another connected via radio or infrared.

Unlike Plug and Play, Universal Plug and Play is built on top of networking and enables ad hoc peer-to-peer connectivity. Networking, in this context, describes a style of connectivity that enables any networked device to initiate a communication with any other networked device, without having established a prior relationship or maintaining a persistent relationship between the devices. Networking also allows multiple devices to establish one or more connections with a single device, and it allows for a device to be capable of both initiating and accepting connections to/from other devices. The PnP, or host/peripheral, model is suitable whenever there is a natural persistent relationship between two devices (e.g. a keyboard, mouse and display maintain a persistent relationship with a host computer). Even though networking does not mandate low level persistent relationships, it provides the needed anchors (addresses) for applications to choose to maintain associations as a convenience for the customer (e.g. remembering commonly used networked printers).

In order to achieve multiple vendor peer-to-peer interoperation among devices, vendors desirably agree on common technology and standards up to the highest level of desired functional interoperation.

UPnP leverages formal protocol contracts to enable peer-to-peer interoperation. Protocols contracts enable real-world multiple-vendor interoperation.

UPnP enables devices to expose a user interface by leveraging browser technology. In this context, the browser can be considered to be a very rich remote terminal. Current browser technology does not maintain a separation of presentation from data, or in the case of devices, control. It is possible to hunt through a page of HTML to extract data values, but it is not convenient or robust. UPnP leverages the separation of presentation and data enabled by the use of XML, and it extends this technology to the device control domain.

UPnP provides a device-driven auto-configuration capability that preserves the experience that customers have on the Web. Today, it is possible to navigate around the Web without loading programs beyond the browser itself. Since UPnP enables the browser to be extended to control devices, and because UPnP devices are controlled with explicit protocols, the browser must somehow learn how to talk to UPnP devices. This learning process is driven entirely from the device itself and is accomplishing entirely by uploading an XML document that describes the capabilities of the device. The architectural component that enables device-driven auto-configuration is called the Rehydrator. The job of the Rehydrator is to convert between APIs and protocols.

Since the auto-configuration process itself is driven only by the exchange of formatted data, there is very little opportunity for a malicious attack from a hostile piece of code.

There are some scenarios where the Web UI model is not sufficient for a rich customer experience. It would not be convenient to have to a separate Web UI for each light switch in a house. To support a rich user interface and to enable the aggregation of devices into a single UI, UPnP enables application control in addition to browser control of devices. This

is achieved simply by enabling applications to call the same Rehydrator APIs that the browser does. Applications can also directly generate and consume the raw UPnP control protocols, provided they are not interested in the device-driven auto-configuration enabled by the Rehydrator.

UPnP assumes that there will be more than one device with UI that wants to control other devices in any given network, and it provides a simple mechanism that enables these control points to remain in sync. This mechanism can easily support device front panels and wireless remotes that do not run UPnP protocols. The UPnP control model is third-party control; any device can transfer bulk data (e.g. files) or A/V data streams from any device on the network, to any device on the network, under the control of any device on the network.

Terminology

The detailed description that follows uses the terminology defined below.

Module. A component of a device, software program, or system that implements some "functionality", which can be embodied as software, hardware, firmware, electronic circuitry, or etc.

User Control Point. The set of modules that enable communication with a UPnP Controlled Device. User Control Points initiate discovery and communication with Controlled Devices, and receive Events from Controlled Devices. User Control Points are typically implemented on devices that have a user interface. This user interface is used to interact with Controlled Devices over the network. The modules minimally include a Discovery Client, a Description Client, a Rehydrator, an Event Subscription Client and an Event Sink. User Control Points may also

include Visual Navigation, a Web browser and an application execution environment. User Control Points can add value to the network by aggregating the control of multiple Controlled Devices (the universal remote) or they can implement a function as simple as initiating the transfer of data to or from a Controlled Device. Examples of devices that could be User Control Points are the personal computer (PC), digital television (DTV), set-top box (STB), handheld computer and smart mobile phone, and the like. Nothing prevents a single device from implementing the functionality of a User Control Point and one or more Controlled Devices at the same time.

Controlled Device. The set of modules that perform certain tasks (e.g., printing) and communicate with a User Control Point. Controlled Devices respond to discovery requests, accept incoming communications from User Control Points and may send Events to User Control Points. Devices that support Controlled Device functionality may also support local user interfaces such as front panel displays or wireless remotes. The modules minimally include a Discovery Server, a Description Server, a Control Server, an Event Subscription Server and an Event Source. Controlled Devices may also include a Presentation (e.g., Web) Server. Examples of devices that could be Controlled Devices are the VCR, DVD player or recorder, heating/ventilation/air-conditioning equipment (HVAC), lighting controller, audio/video/imaging playback device, handheld computer, smart mobile phone and the PC, and the like. Nothing prevents a single device from implementing the functionality of a User Control Point and one or more Controlled Devices at the same time.

Bridge. A set of modules that enables Bridged and Legacy Devices to interact with native UPnP devices. The bridge itself exposes a collection of UPnP Controlled Devices to User Control Points. The Bridge maps between native UPnP Device Control Protocols and the underlying protocols or other control methods exposed by the Bridged and Legacy Devices. Optionally, such a device could expose UPnP Controlled Devices to Legacy Devices in the manner required by the Legacy Devices. Nothing prevents a single device from implementing the functionality of a User Control Point, one or more Controlled Devices and a Bridge at the same time.

Service Provider. A module used by a UPnP Bridge that translates between UPnP protocols and the protocols used by Bridged and Legacy Devices. No Service Providers are required for communication among native UPnP devices.

Bridged Device. A device that cannot participate in UPnP at the native protocol level, either because the device does not have sufficient resources or because the underlying media is unsuitable to run TCP and HTTP. Examples of devices that could be Bridged Devices are power line-controlled A/V equipment, light switches, thermostats, wristwatches and inexpensive toys. Bridged Devices are UPnP complaint and are exposed to other UPnP devices through a UPnP Bridge.

Legacy Device. Any non-UPnP compliant device that must be exposed to other UPnP devices through a UPnP Bridge.

Device Model. The UPnP model of Controlled Devices. The Device Model includes the addressing schemes, Description Document, Devices and Services hierarchy and the functional description of Services.

Device Control Protocol (DCP). A complete set of UPnP protocols and schemas used to interact with a UPnP Controlled Device.

Device Definition. The formal definition of a Device Type. A Device Definition includes a Device Type Identifier, the fixed elements in the Description Document, the required set of Service Definitions in the Root Device, and the hierarchy of required Devices and Service Definitions.

Service Definition. The formal definition of a Service Type. A Service Definition includes a Service Type Identifier, definition of the Service State Table (SST), definition of the Service Command Set, the Service Control Protocol (SCP) and Service Control Protocol Declaration (SCPD).

Device. In the context of the Device Model, a container for Services. A Device generally models a physical entity such as a VCR, but can also represent a logical entity. A PC emulating the traditional functions of a VCR would be an example of a logical device. Devices can contain other Devices. An example would be a TV/VCR packaged into a single physical unit. UPnP enables the association of user interface (display icon and root Web page) with every Device, including Root Device.

Root Device. The topmost Device in a hierarchy of nested Devices. A Device with no nested Devices is always a Root Device.

Device Type. A relatively high level classification of Devices with common functionality. Device Type is intended to enable Devices to be simply and automatically grouped for search and/or presentation. An example of a Device Type is "VCR". Device Types are formally defined in terms of a required set of Service Definitions of minimum version that a

compliant Device must support. UPnP supports searches for all Devices of a specified Device Type.

Device Type Identifier. A unique identifier that identifies a Device Definition. This identifier adheres to the format of a Uniform Resource Identifier (URI). See, T. Berners-Lee, R. Fielding, L. Masinter, Uniform Resource Identifiers (URI): Generic Syntax, IETF RFC 2396 (August 1998).

Device Friendly Name. A human readable string that is usually initialized by vendors at the time of manufacturer of a Device. Every Device, including Root Devices, has a Device Friendly Name. A typical Device Friendly Name will contain manufacturer and model information, and especially when interpreted by humans, can be used to enable a more precise identification of a UPnP Device from the set of discovered Devices. Once identified, the Unique Device Name (UDN) can be used to unambiguously identify the same Device in the future. UPnP enables Device Friendly Names to be changed by User Control Points. The Device Friendly Name should not be used as device identifier.

Unique Device Name (UDN). The fundamental identifier of a Device. Every Device, including Root Devices, has exactly one UDN. The UDN is globally unique and permanent, even across power cycles and physical location changes. The UDN is the only UPnP device identifier guaranteed never to change. UPnP enables searches for devices by UDN.

Description Document. A structured unit of data that is used by a User Control Point or UPnP Bridge to learn the capabilities of a Controlled Device. Description Documents are retrieved from the Description Server on a UPnP Controlled Device. There is one Description Document for every

Root Device that describes the Root Device and all non-Root Devices. Description Documents adhere to XML grammar. To support localization, multiple Description Documents can exist. A User Control Point requests the preferred localized Description Document by using the standard HTTP "accept-language" header.

Service. The fundamental UPnP controllable entity (but not the finest level of control). An example of a Service is "Clock". Services are defined with a mandatory common base set of functionality. Vendors can extend the base set with proprietary extensions provided the base functionality is implemented. Service Definitions are versioned and later versions are constrained to be supersets of previous versions. UPnP enables searches for all Devices that contain a specified Service of a minimum version. This search would find all clocks, regardless of their packaging. A search for Device Type "Clock" would be used to find only stand-alone clocks.

Service Type. A classification of Services by their function.

Service Type Identifier. A unique identifier that identifies a Service Definition. This identifier adheres to the format of a Uniform Resource Identifier (URI). See, T. Berners-Lee, R. Fielding, L. Masinter, Uniform Resource Identifiers (URI): Generic Syntax, IETF RFC 2396 (August 1998).

Service State Table (SST). A logical table consisting of rows of [*Variable, Type, Legal Values, Default Value, Current Value*] that represents the current electrical, mechanical and/or logical state of a Service. SST instances are stored on the Controlled Device itself and are the ultimate authority of the state of the Service. All local user interface, such as front

panels or wireless remotes are required to update the SST on UPnP compliant devices.

SST Definition:

Service Command Set. A set of Commands that can be invoked on a Service. Commands generally result in changes in the Current Value field of one or more rows of a SST. Commands are logically represented in the format *Command* (*Variable* = *New Value*, *Variable* = *New Value*, ...). Services must accept or reject the complete set of changes to a SST. There is a mandatory standard Query Command that is used to retrieve the Current Value of any row of a SST.

Service Command Set Definition:

Service Control Protocol (SCP). The protocol used to invoke Commands against a Service and to return results. There is exactly one SCP per Service Definition. SCPs adhere to the grammar of SCP XML schema. SCPs can be generated by an automated tool that accepts a SST Definition and a Command Set Definition as input.

Service Control Protocol Declaration (SCPD). A formal representation of the schema of a Service. The SCPD declares the rows of a Service's SST and the associated Command Set. SCPDs are uploaded from Controlling Devices in their Description Documents and enable User Control Points or Bridges to invoke Commands on the Service without any prior or persistent knowledge of the capabilities (or schema) of the Service. There is exactly one SCPD per Service Definition. SCPDs adhere to XML grammar. SCPDs can be generated by an automated tool that accepts a SST Definition and a Command Set Definition as input.

Event. An unsolicited message generated by a Controlled Device and delivered to one or more User Control Points. Events are used to maintain a consistent view of the state of Service across all interested User Control Points. UPnP leverages the GENA event architecture (see "Generic Event Notification") to transport event messages. All events are delivered using TCP/IP for reliability.

Generic Event Notification Architecture (GENA). An event transport protocol. GENA leverages TCP/HTTP as a transport. GENA has been submitted as an Internet Draft to the IETF. See, J. Cohen, S. Aggarwal, Y. Goland, General Event Notification Architecture Base: Client to Arbiter, IETF Internet Draft, "draft-cohen-gena-client-00.txt."

Simple Service Discovery Protocol (SSDP). A simple network device discovery protocol. UPnP uses SSDP to allow User Control Points to find Controlled Devices and Services. SSDP operates in a default, completely automatic multicast UDP/IP based mode in addition to a server-based mode that uses TCP/IP for registrations and query. Transitions between the default dynamic mode and server-based mode are automatic and transparent to upper level software. SSDP enables every Controlled Device to control the lifetime that its Description URL is cached in all User Control Points. This enables a Controlled Device to remain visible to User Control Points for a relatively long time (through power cycles), in addition to enabling a Controlled Device to appear and disappear very quickly, all under the control of the Controlled Device. SSDP and related Multicast and Unicast UDP HTTP Messages specifications have been submitted as Internet Drafts to the IETF. See, Y. Goland, Multicast and Unicast UDP HTTP Messages, IETF Internet Draft, "draft-goland-http-udp-00.txt;" and Y.

Goland, T. Cai, P. Leach., Y. Gu, S. Albright, Simple Service Discovery Protocol/1.0, IETF Internet Draft, "draft-cai-ssdp-v1-02.txt."

Client. In the context of UPnP, Client refers to a module that initiates a TCP/HTTP connection to a peer HTTP server.

Server. In the context of UPnP, Server refers to an HTTP server. This is a module that accepts incoming TCP/HTTP connections and either returns a Web page or forwards the payload data to another module. Client and Server describe only the direction of initiation of TCP/HTTP connections. There is no relationship between the low level concepts of Client and Server and the high level concepts of User Control Point and Controlled Devices. Logically, User Control Points always discover and initiate communication with Controlled Devices, but this communication requires Client and Server functionality on both sides.

Hostname. A Hostname is the Domain Name System (DNS) or NetBIOS Name Service (NBNS) that, when resolved to an IP address, represents a network interface that can be used to establish TCP/IP level connectivity to User Control Points, Controlled Devices or Bridges. Hostnames can be used to provide persistent network level addressing on a network where IP addresses are dynamically assigned and of unknown lifespan or to integrate with an existing managed network. UPnP provides an algorithm for seeding a device's hostname from its UDN at manufacturing time.

Uniform Resource Locator (URL). A format for expressing Web addresses. URLs minimally contain an identification of the protocol family that the URL is valid for, a Hostname, and a path. UPnP uses URLs as

addresses whenever the module accepting the incoming connection is an HTTP server.

Description URL. The URL returned from a Controlled Device or Bridge in response to any UPnP SSDP query. This URL always points to a Description Server on the Controlled Device. An HTTP GET can be issued on this URL to retrieve the Description Document. This URL is valid as an address for the lifetime of the Hostname embedded in the URL.

Discovery Server. The module that runs in a Controlled Device or Bridge that responds to SSDP queries. This Server is unique in that it must support UDP/HTTP in addition to TCP/HTTP.

Discovery Client. The module that runs in a User Control Point that initiates SSDP queries.

Description Server. The module that runs in a Controlled Device or Bridge that responds to HTTP GETs and returns Description Documents. This service consists of a TCP/HTTP server than can retrieve and return a Description Document from persistent storage (like a filesystem).

Visual Navigation. User Control Point functionality that displays the icons of discovered Devices and enables the transfer of control to a browser or application to interact with the Controlled Device. In Windows, Visual Navigation could be implemented as a folder of icons.

Presentation URL. A URL that can be used by a User Control Point to navigate to the Presentation Server of a Controlled Device. This URL is returned in the Description Document and is valid as an address for the lifetime of the Hostname embedded in the URL. All Devices, including non-Root Devices, can have an associated Presentation URL.

Presentation Server. A Web Server in most common cases. The module that runs in a Controlled Device that responds to HTTP GETs or Presentation URLs and returns user interface using Web technologies (JavaScript, Jscript®, ECMAScript, VBScript, ActiveX®, Java Applet, etc.).

Browser. A Presentation Client. A Web browser extended with a Rehydrator.

Control URL. A URL that can be used by a User Control Point to navigate to the Control Server of a Controlled Device or Bridge. This URL is returned in the Description Document and is valid as an address for the lifetime of the Hostname embedded in the URL. All Services have an associated Control URL.

Control Server. The module that runs in a Controlled Device or Bridge that responds to Commands invoked on a Service by a User Control Point. Commands are encoded and sent using the SCP specified in the Service Definition. This service consists of a TCP/HTTP server that passes control to the native control logic of a Service, updates the SST and generates an event if the SST changes.

Rehydrator. In UPnP, a Control Client. A User Control Point module that translates between native operating system APIs and SCPs and events. The Rehydrator uploads SCPDs from Controlled Devices and Bridges and generates appropriate SCPs in response to application API requests to invoke Commands.

Event Subscription URL. A URL that can be used by a User Control Point to navigate to the Event Subscription Server of a Controlled Device or Bridge. This URL is returned in the Description Document and is valid as an

address for the lifetime of the Hostname embedded in the URL. All Services have an associated Event Subscription URL.

Event Subscription Server. The module that runs in a Controlled Device or Bridge that responds to GENA SUBSCRIBE requests from User Control Points. A SUBSCRIBE informs the Controlled Device or Bridge of the User Control Point's desire to receive future events. This service consists of a TCP/HTTP server that adds the User Control Point's Event Sink URL to the list of destinations to be NOTIFY'd whenever the SST associated with the Service changes.

Event Subscription Client. The module that runs in a User Control Point that sends GENA SUBSCIBE messages to the Event Subscription Server.

Event Sink URL. A URL, supplied by a User Control Point, that is used as an address to send event NOTIFYs to. This URL is valid as an address for the lifetime of the Hostname embedded in the URL. There is no explicit relationship between Event Sink URLs and Subscription Identifiers.

Subscription Identifier (SID). A header in the GENA NOTIFY message that identifies the source of an event. In UPnP, the SID can be considered as an alias for the Event Source instance.

Event Sink. The module that runs in a User Control Point that accepts incoming GENA event NOTIFYs. This service consists of a TCP/HTTP server that passes the event information to interested applications running on the User Control Point.

Event Source. The module that runs in a Controlled Device or Bridge that sends GENA NOTIFYs to the Event Sink Servers of SUBSCRIBES User Control Points.

Domain Name System (DNS). A distributed system of servers that locates the IP addresses of other computers on a network based on their hierarchical names.

NetBIOS Name Server (NBNS). A server that locates the IP addresses of other computers on a network based on their flat NetBIOS computer names.

Multicast DNS (MDNS). A peer-to-peer translation scheme that does not require involvement of DNS servers.

UPnP Technologies Overview

An overview of technologies utilized in UPnP follows.

Device Discovery: Simple Service Discovery Protocol (SSDP)

TCP/IP provides the ability to initiate a connection with a specified application running on a specific device, provided both the network address of the device (IP address) and the application address (port) are known. Generally, application addresses (ports) are standardized and widely known, but the problem of learning the IP address of a device remains.

Simple Service Discovery Protocol (SSDP) is a protocol that enables devices to learn of the existence of potential peer devices and the required information (an IP address) needed to establish TCP/IP connections to them. The successful result of an SSDP search is a Uniform Resource Locator (URL). The Hostname embedded in the URL can be resolved to an

IP address that can be used to make a connection to the discovered device. The name to address resolution is outside of the functionality of SSDP.

SSDP specifies a default, completely automatic, best-effort multicast UDP-based operating mode, in addition to a server mode that uses TCP for registration and query. Fall-forward to server mode and fallback to the default dynamic mode can occur automatically and transparently as a server is added or removed from a network. Server mode can be used to reduce network traffic, to implement searches based on location or policy and to integrate with a directory system.

SSDP requires that all devices specify a maximum lifetime that SSDP level knowledge of the device will remain cached in other network devices. If a device does not refresh the cache of other network devices before this interval expires, the device will be assumed to have disappeared from the network. This interval can be chosen to be larger than a typical power down cycle to enable device visibility to persist for a relatively long time, or a smaller interval can be chosen to enable more dynamic visibility control. In all cases, devices that are abruptly removed from the network will eventually disappear from all networked devices.

In response to an SSDP search, UPnP devices return a Description URL in the SSDP Location and optionally the Alternate Location (AL) SSDP headers. An example location header is a follows:

Location: http://device.local/description/path/description.xml

In this example, the device.local is the Hostname of the Controlled Device, and the "description/path/description.xml" element of the URL is the path and name of the Description Document on the device.

Eventing: Generic Eventing Notification (GENA)

Eventing, in the context of UPnP, is the ability for a device to initiate a connection at any time to one or more devices that have expressed a desire to receive events from the source device. Events are used to enable synchronization among multiple devices organized into a many to one relationship. UPnP events are mainly used for asynchronous notifications of state changes.

TCP/IP provides the fundamental support for the connections that carry event information reliably. Generic Event Notification (GENA) adds conventions for establishing relationships between interested devices and an addressing scheme to enable the unambiguous delivery of events. GENA leverages HTTP addressing and encapsulation.

User Control Points, Controlled Devices, Bridges and OS Service Remoter/Publisher

With reference now to Figures 1 and 2, UPnP is an application-level distributed network architecture where the logical nodes on the network are User Control Points 104-105, Controlled Devices 106-107, Bridges 120, and OS Service Remoter/Publisher 150. These classifications refer to functionality rather than physical entities. The functionality of UPnP User Control Points 104-105, Controlled Devices 106-107 and Bridges 120 can be packaged into physical entities (e.g., multiple function devices 102-103) in any combination.

The primary distinction between a User Control Point 104-105 and a Controlled Device 106-107 is that the User Control Point is always the communication initiator. After the initial communication, User Control Points can receive events from Controlled Devices.

Controlled Devices 106-107 are responsible for storing and updating the state of Services. User Control Points are required to synchronize to the state on Controlled Devices and to share state directly among themselves.

User Control Points typically have user interface that is used to access one or more Controlled Devices on the network. Controlled Devices typically only have local user interfaces.

Bridges 120 (Figure 2) expose devices that do not expose native UPnP protocols as native UPnP Controlled Devices. The Bridge itself looks to other UPnP User Control Points like a set of Controlled Devices.

Service Remoter/Publisher 150 (Figure 2) exposes services 156-157 of a general purpose operating system 152 (e.g., services provided in the Microsoft Windows operating system), hereafter referred to as OS services. The Service Remoter/Publisher 150 emulates a set of Controlled Devices providing the operational functionality of the OS services, and thus appears as a set of such Controlled Devices to the UPnP User Control Points 104-105. The Service Remoter/Publisher 150 then interacts with the OS services through the conventional OS API exposed by the OS services in response to the UPnP User Control Points 104-105 to effect remote control of the OS services from the UPnP User Control Points via the UPnP protocol.

UPnP Publishing Service and Service Remoter

With reference to Figure 3, the Service Remoter/Publisher 150 (Figure 2) is implemented in an exemplary general purpose operating system 152, which may be for example the Microsoft Windows operating system, as a UPnP Publishing Service 160 and a Service Remoter 162.

The general purpose operating system 152 provides a variety of OS services 156 that perform specific system functions to support other programs, including application programs run on the operating system. The illustrated OS services 156 include date and time clock, file and printer sharing, print spooling, electronic mail, fax, telephony, file synchronization (e.g., with mobile devices), file backup and archiving, audio and video file streaming (e.g., from local or remote storage, and to a peripheral play-back device, such as speakers or a monitor), audio and video codecs, file compression/decompression, problem diagnosis and support, event logging, user and software configuration administration, security and access control, remote access (RAS), networking (Internet socket, FTP, HTTP, TCP/IP, NetBIOS, etc.), name and address (DHCP, DNS), remote procedure call (RPC) and named pipe, dynamic peripheral configuration (Plug and Play), and application scheduling (e.g., Windows 95 system agent), among others. In addition, the OS services can include extended services that plug into an extendible services architecture of the operating system, such as using the Microsoft Windows NT Services programming model (see, for example, J. Richter, Design A Windows NT Service To Exploit Special Operating System Facilities, Microsoft Systems Journal (October 1997). The OS services 156 are exposed to programmatic control by applications and other programs running on the operating system through a set of OS APIs 166 (e.g., the Microsoft Win32 API, which is described in Microsoft Win32 Programmer's Reference, Microsoft Press (1993).)

The UPnP Publishing Service 160 in the illustrated general purpose operating system 152 is a program structured as a service of that operating system (e.g., as a Windows NT Service), and provides a publishing API 164. The UPnP Publishing Service 160 operates as a general mechanism

for publishing the information necessary to emulate the functionality of devices and services running under the general purpose operating system as Controlled Devices per the UPnP protocol for devices and services that do not themselves support the UPnP protocol. The Service Remoter 162, the Bridge 120, as well as Applications 170-171 use the publishing API 164 to establish a Controlled Device emulation of their functionality by the UPnP Publishing Service 160, and provide the communications (such as of the service state and Commands from User Control Points) between the programs and the UPnP Publishing Service to effect the emulation as Controlled Devices. The UPnP Publishing Service 160 operates to publish information such as discovery responses, service description, and service state, as well as direct responses to UPnP requests, to User Control Points 104 for the set of Controlled Devices that it emulates. The UPnP Publishing Service 160 also passes information back to the devices and services running under the general purpose operating system (e.g., via a callback registered with the UPnP Publishing Service through the publishing API) to effect the Commands from the User Control Point 104 to the emulated Controlled Devices. Thus, for purposes of the User Control Points interacting with the emulated Controlled Devices in accordance with the UPnP protocol, the UPnP Publishing Service appears as the set of emulated Controlled Devices.

The UPnP Publishing Service and publishing API allows the Service Remoter 162, the Bridge 120 and applications 170-171 to expose their functionality for remote control though the UPnP protocol without having to individually implement all the various components (described below and illustrated in Figure 5) of a Controlled Device per the UPnP protocol. Instead, the UPnP Publishing Service implements the components of a

Controlled Device collectively for the set of emulated Controlled Devices. The Service Remoter 162, the Bridge 120 and applications 170-171 need only be programmed to use the publishing API.

The Service Remoter 162 is a program module that operates to convert between the specific OS API of the particular OS services, and the publishing API of the UPnP Publishing Service 160. The Service Remoter 162 uses the publishing API to effect emulation of the particular OS Services as UPnP Controlled Devices by the UPnP Publishing Service 160, which exposes the OS services' operational functionality to remote control by User Control Points via the UPnP protocol. Also, as information is passed back from the UPnP Publishing Service 160, such as for Commands from the User Control Points in the UPnP protocol intended to control the OS services' operation, the Service Remoter 162 uses the regular OS API of the OS services to effect control of the OS services. This avoids having to restructure or reprogram the OS services as UPnP Controlled Devices, or even to themselves use the publishing API of the UPnP Publishing Service 160.

Figure 4 illustrates an exemplary implementation of the UPnP Publisher/Remoter 150 (Figure 2) in the Microsoft Windows operating system. In this exemplary implementation, the UPnP Publishing Service 160 includes an instance of the Service Remoter 162 per each OS service 156 that is to be exposed as a UPnP Controlled Device, and a single SSDP Service 186 shared by all remoted operating system services 156 on the system. The Service Remoter 162 in this exemplary implementation fully implements a UPnP controlled device that remotely exposes the functionality of the respective OS service 156. The Service Remoter 162 includes a set of library modules 190-198 that implement the various

discovery, description, eventing and control protocols of UPnP, as well as mapping code for converting communications with UPnP user control points per the UPnP protocol into interactions via the OS API with the OS service 156. These protocol modules includes a UPnP Publishing API library 190, a SSDP server library 191, an HTTP server library 192, a Sockets library 193, a GENA server library 194, a SOAP server library 195, a GENA publisher library 196, an HTTP client library 198, and an XML client library 197.

The SSDP service 186 provides support for the SSDP discovery protocol for each OS service remoted through the UPnP Publishing Service 160. The SSDP service 186 includes a separate instance of the Sockets library module 189 and a publish list 188. Each OS service 156 to be remoted through the UPnP Publishing Service 160 is added to the publish list 188 of the SSDP service 186 via a call to the UPnP Publishing API library 190 and SSDP server library 191. The SSDP Service 186 then provides appropriate responses to SSDP discovery requests per the UPnP protocol for the remoted OS service 156.

The Service Remoter 162 handles incoming UPnP description requests and responses for the remoted OS service 156 via the Sockets library 193, the HTTP server library 192, and the UPnP Publishing API library 190 modules.

The Service Remoter 162 handles incoming event subscribe requests for the remoted OS service 156 via the Sockets library 193, the HTTP server library 192, and the UPnP Publishing API library 190 modules. Data for out-going event publishing for the remoted OS service 156 (e.g., the service's state) is handled in the Service Remoter 162 through the UPnP

Publishing API library 190, the GENA publish library 196, the HTTP client library 198 and the Sockets library 193 modules.

The Service Remoter 162 further handles control commands of UPnP User Control Points to the remoted OS service 156 via the Sockets library 193, the HTTP library 192, the SOAP server library 195, and the UPnP Publishing API library 190 modules.

With reference now to Figure 5, the following table lists the modules in the User Control Points 104-105 and Controlled Devices 106-107, along with their functions.

User Control Point		Controlled Device	
Function	**Module**	**Function**	**Module**
Initiate discovery of Controlled Devices.	Discovery Client	Respond to discovery requests.	Discovery Server
Retrieve Description Documents.	Description Client	Provide Description Documents.	Description Server
Display a folder of icons per discovered Device and allow transfer of control to a selected device.	Visual Navigation		

View user interface exposed by a Controlled Device.	Web Browser	Provide user inteface for remote User Control Points.	Presentation (Web) Server
Execute applications.	Application Execution Environment		
Invoke Commands on a Controlled Device by sending Service Control Protocols in response to local API calls.	Rehydrator	Accept incoming Commands in SCPs and execute them.	Control Server plus native control logic
Inform a Controlled Device of a desire to receive Events.	Event Subscription Client	Accept requests for Events and remember them.	Event Subscription Server
Receive an Event.	Event Sink	Send an Event.	Event Source

Device Model

The UPnP Device Model 200 shown in Figure 5 is the model of a UPnP Controlled Device or Bridge that is emulating native Controlled Devices. The Device Model includes the addressing scheme, eventing

scheme, Description Document schema, Devices and Services schema and hierarchy, and the functional description of modules. The UPnP Device Model extends beyond simple API or a command and control protocol definitions to enable multiple User Control Points to have a consistent view of Controlled Devices. This requires that the state of running services be formally modeled and that all state changes be visible to User Control Points. Central to the distributed UPnP architecture is the rule that Controlled Devices are the ultimate authority for the state of Services running on them.

UPnP Service

The fundamental controllable entity in UPnP is a UPnP Service 210-217. Every running instance of a UPnP Service includes:

- A Service State Table (SST) 230, which represents the current state of the UPnP Service.

The SST 230 can be used to represent the operational mode of device or to act as an information source or sink for structured data or simple files. The SST of a VCR 254 (Figure 6) could represent the current transport mode, tuner channel selection, input and output switch selections, audio and video decoding format and current timer program. The SST of clock 251 (Figure 6) would likely represent the current time. The SST of an image rendering device could implement a video frame-buffer that can accept raw pixel information or formatted JPG files. The SST of an audio or video playback device could implement a transfer buffer or queue of material to be played. The SST of PDA could implement a collection of formatted data that has changed

and needed to be synchronized with another device, in addition to a transfer buffer for accepting incoming formatted data.

The logical structure of a SST published in the Service Definition, but the actual storage format of an instance of a SST is entirely up the device. The only interaction with a SST is through a formal application level network protocol.

- A Control Server 232, which accepts incoming Commands expressed in the UPnP Service's Service Control Protocol (SCP). The Control Server passes the command to the UPnP Service's native command processing logic and waits for command completion. When the command is completed successfully, the SST is updated, an event is generated, and a successful response is returned to the User Control Point. In the event of an illegal command or unsuccessful command, no changes are made to the SST and a failure response is returned. The Command and response sequence is payload to a TCP/HTTP request/response.
- An Event Subscription Server and Event Source 234. The Event Subscription Server accepts incoming GENA SUBSCRIBE messages from User Control Points and adds them to a list of User Control Points interested in SST change events from the UPnP Service. The Event Source initiates a TCP/HTTP connection to each interested User Control Point and sends a GENA NOTIFY each time the UPnP Service's DST changes. The NOTIFY payload includes the changed contents of the DST.
- A Control URL that identifies the Control Server.
- An Event URL that identifies the Event Subscription Server.

The formal definition of a UPnP Service (Service Definition) includes:

- The definition of the SST. SST layouts are logically specified in terms of rows of [*Variable, Type, Legal Values, Default Value*]. The actual instance of a SST would also include a *Current Value* field in every row.
- The definition of the Service Command Set that can be invoked against the UPnP Service's SST. Commands are logically specified in terms of *Command* (*Variable = New Value, Variable = New Value, ...*). If a Command results in more than a single Variable change, the updates are atomic and the Command will fail if it is illegal to make the specified change to any one Variable.
- The definition of a structured unit of data called a Service Control Protocol Declaration (SCPD). SCPD is used to advertise the layout (schema) of the SST and Command Set of the UPnP Service to a User Control Point or Bridge. The SCPD enables the User Control Point to invoke Commands (through the Rehydrator) on the Controlled Device without any prior or persistent knowledge of the capabilities of the device. The SCPD is uploaded from the Controlling Device as part of the Description Document. Generation of the SCPD for a UPnP Service based on its SST definition and Command Set definition can be fully automated.
- The definition of a network protocol used to invoke Commands against the SST associated with a UPnP Service and to return results. The SCP can be generated from the SCPD. The Rehydrator's job is to convert SCPDs into SCPs. The reason for a formal SCP specification is to enable the implementation of the Control Server itself and to enable simple peer-to-peer device interoperation using only published protocols.

- An identifier, called the Service Type Identifier, that identifies a unique Service Definition. Service Definitions are versioned in controlled manner. Every later version of a UPnP Service must be proper superset of the previous version.

Device

According to the device model 200 shown in Figure 5, a UPnP Device 202-205 (e.g., multiple function devices 102-103 of Figure 1 and bridged devices 122-123 of Figure 2) is a logical container of one or more UPnP Services 210-217. Generally a Device represents a physical entity such as a VCR. Typical UPnP Services in the VCR Device example might be "TRANSPORT", "TUNER", "TIMER" and "CLOCK". While Devices are often physical entities, a PC emulating the traditional functions of a VCR could also be modeled in the same way as the stand-alone VCR. Devices can contain other Devices. An example would be a TV/VCR 250 (Figure 6) packaged into a single physical unit. A Device (e.g., devices 202-203) may also be a logical container of other Devices. The top-most Device in a hierarchy of nested Devices 203-205 is called the Root Device 202. A Device with no nested Devices is always a Root Device.

The UPnP Device Model was designed to be general and flexible. It should be possible to model an entire Nuclear Power Plant as a single UPnP Service or as a deeply nested hierarchy of Devices and UPnP Services. In general, a UPnP Service 210-217 is cohesive set of functions that enables flexible packaging into a variety of Devices. UPnP Services can be versioned independently of Devices.

All Devices, including Root Devices belong to one or more Device Types. Device Types are intended to enable instances of Devices to be

simply and automatically grouped for presentation. An example of a Device Type is "VCR" 254 (Figure 6). Device Types are formally defined in terms of a minimal set of versioned UPnP Services that a Device of *Device Type* must support. Device Types are not formally versioned. Device Type is a relatively high level grouping. A Device of *Device Type* only ensures that minimal set of UPnP Services of a minimal version is present. There can be other UPnP Services, higher versioned UPnP Services and UPnP Services with vendor extensions present on such a Device.

UPnP enables SSDP level searches for a unique instance of a Device (by UDN), all Devices of type *Device Type* and all Devices that contain at least one Service Type of minimum version. The result of an SSDP search is always a URL that points to the Description Document contained in the Root Device. In the event that matching Device is not the Root Device, the Description Document has a tree of nested Devices that can be traversed to find the matching Device.

Every Device includes:

- One or more Device Types.
- One or more UPnP Services.
- Optionally, one or more Devices.
- Optionally, a Presentation (Web) Server 220-223 that can be used to expose Device user interface. Every Presentation Server has an associated Presentation URL.
- A globally unique identifier called the Unique Device Name (UDN). The UDN is the fundamental identifier of an instance of a Device. Every Device, including Root Devices, has exactly one UDN.

Every Root Device 202 also includes the Description Document 226 and Description Server 228 for all Devices under and including itself.

The formal definition of a Device (Device Definition 226) includes:

- The fixed elements of the Description Document that describe the Device.
- The required hierarchy of Devices and Service Definitions.

There can be many Device Definitions that belong to a single Device Type.

Device Types

The formal definition of a Device Type includes:

- A Device Type Identifier.
- The required hierarchy of Devices and Service Definitions of minimum versions.

Service State Table

A Service State Table (SST) logically consists of rows of:

Variable, Type, Legal Values, Default Value, Current Value

Although entries of the Service State Table in UPnP consist of these five items, the state table alternatively can contain fewer or additional items. Generally, each entry will minimally consist of a Variable name or identifier, and its current value.

The following table lists various Types available in UPnP.

Type	Description	Example
String	A sequence of UNICODE characters.	
Number	A number, with no limit on digits; may potentially have a leading sign, fractional digits, and optionally an exponent. Punctuation as in US English.	15, 3.14, -123.456E+10
Boolean	TRUE or FALSE.	
DateTime	A date in ISO8601 format, with optional time and optional zone. Fractional seconds may be as precise as nanoseconds. See, Data Elements And Interchange Formats – Information Interchange – Representation Of Dates And Times, International Standard, ISO 8601, First Edition 1988-06-15.	19941105T08:15:5+03
ByteBlock	An unstructured sequence of bytes.	

The ByteBlock is essentially a data buffer. In one use, a variable of this type can be used to effect transfer of a file from the Controlled Device to the User Control Point. The file to be transferred is kept in the Service State Table as the current value of this variable. On a change in the file,

the file is transferred to any subscribing User Control Point in an event notification.

The reason for representing UPnP Services this way is to ensure that the state of a UPnP Service is easily available in a common way to multiple User Control Points.

An SST can be used to represent to current operational mode of device, act as an information source or sink and/or simply be a repository for commands. The SST of a VCR UPnP Service could represent the current transport mode, tuner channel selection, input and output switch selections, audio and video decoding format and current timer program. Alternatively, the VCR 254 could be represented as a Transport UPnP Service 260, Tuner UPnP Service, I/O Switch UPnP Service, A/V Decoding Configuration UPnP Service and Programmable Timer UPnP Service 261.

The SST of a clock 251 would likely represent the current time. Additionally an alarm clock could include Service Variables to configure the clock.

The SST of an image rendering device could implement a video frame-buffer that can accept raw pixel information or formatted JPG files. The SST of an audio or video playback device could implement a transfer buffer or queue of material to be played. The SST of PDA could implement a collection of formatted data that has changed and needed to be synchronized with another device, in addition to a transfer buffer for accepting incoming formatted data.

User Control Point Synchronization

In accordance with an device state and eventing model illustrated in Figure 7, UPnP rules require that every change to an SST generate a

corresponding event to announce the change to the all interested User Control Points.

Device Addressing

With reference now to Figure 8, UPnP is built on top of HTTP and leverages the native address format of the Web, Uniform Resource Locators (URLs). URLs minimally contain an identification of the application protocol family ("http") that the URL is valid for, a Hostname and a path. In the context of UPnP, the path part of a URL can represent either a filesystem path or simply an identifier of the local system module and context that can process incoming messages.

While UPnP modules are described as HTTP servers, there is no requirement that implementations be based on actual Web servers. In most cases, the job of the HTTP server is simply to accept the incoming connection, look at the local destination part of the address (the path) and forward the payload to another module. UPnP enables, but does not require, that all HTTP Servers be based on a common software implementation or runtime instance. Controlled Devices and Bridges can include a TCP port specification as part of a URL to override the default value of 80.

The successful result of a SSDP level search in UPnP is always one or more Description URLs. These URLs can be used to navigate to the Description Document of a Controlled Device or Bridge. A User Control Point uploads the Description Document and extracts the URLs of the Servers running on the Controlled Device or Bridge.

All URLs returned in the Description Document have a lifetime equal to the lifetime of the Hostname embedded in them. User Control Points

can store these URLs as addresses without going through a search sequence first. Once they have been advertised in a Description Document, Controlled Device and Bridges cannot arbitrarily change Server URLs.

Whenever a Hostname changes, all URLs associated with all Devices addressed by that Hostname are invalidated. The UDN is the only UPnP identifier guaranteed never to change. Any persistent associations maintained by applications should at least store the UDN to able to unambiguously identify the target Device.

The lifetime of a Description URL is determined by Controlled Device or Bridge that advertises it. If a Controlled Device or Bridge allows an SSDP advertisement of a Description URL to expire, the URL is invalidated.

User Control Points use the Event Subscription URL returned by the Controlled Device or Bridge to connect to the Event Subscription Server. This server does the housekeeping of remembering all User Control Points that are interested in receiving Events on a UPnP Service. The Event Subscription Server needs an address to send the events back to. This address is called the Event Sink URL, and is supplied to the Controlled Device or Bridge in the GENA SUBSCRIBE message. The lifetime of an event subscription, and the Event Sink URL, is determined by the timeout on the SUBSCRIBE message.

Further details of UPnP addressing are listed in the following table.

UPnP Addresses

URL	Function
Description URL	Points to the Description Server and Document path on a Root Device. This URL is returned by the Description Server as part of the discovery process.
Presentation URL	Points to a Presentation (Web) Server on a Controlled Device. There is one Presentation URL per Device, including Root Devices. This URL can be entered into the address bar of a Web browser to navigate to the root Web page of a Device. This URL is returned in the Description Document.
Control URL	Points to the Control Server implementing a UPnP Service on a Controlled Device. There is one Control URL per instance of a UPnP Service. This URL is returned in the Description Document.
Event Subscription URL	Points to an Event Subscription Server on a Controlled Device. This URL is returned in the Description Document.
Event Sink URL	Points to an Event Sink (an HTTP Server) on a User Control Point. This URL is specified by the User Control Point in the GENA SUBSCIBE message.

Device Discovery and Identification

UPnP enables SSDP searches for a unique Root or non-Root Device by UDN, devices of a specified Device Type and devices containing a UPnP Service of a specified Service Type.

5

UPnP SSDP Level Searches and Results

Search for	Returns
A unique Root Device (by UDN)	A single Description URL pointing to the Description Server and Document path on the Root Device.
A unique non-Root Device (by UDN)	A single Description URL pointing to the Description Server and Document path on the Root Device that contains the non-Root Device.
Type of Device	A set of Description URLs pointing to the Description Servers/Document paths of all Root Devices that match the Device Type, or contain a non-Root Device that matches the Device Type.
Type of Service	A set of Description URLs pointing to the Description Servers/Document paths of all Root Devices that contain a matching UPnP Service, or contain a non-Root Device that contains a matching UPnP Service.

SSDP specifies Service Type (ST), Notification type (NT), and Unique Service Name (USN) header fields for queries and for announcements.

UPnP uses the ST or NT header to carry one of the UPnP defined identifiers. A unique USN is required for each unique SSDP announcement.

Multiple instances of the same Service Type within a Controlled Device 106-107 or Bridge 120 are not independently announced.

UPnP search identifiers are used during the discovery process. The result of a successful discovery is one or more Description URLs. The format for search identifiers is:

```
upnp:searchtype:[ allformat | UDNformat |
srvformat | devformat ]

searchtype      = [ UDN | SrvType | DevType | all
]

allformat       = all

UDNformat = UDN:namespace:uniqueid
namespace = [ GUID | IEEEMAC | 1394]

srvformat = SrvType:servicetype:version
devformat = DevType:devicetype
```

UPnP Search Identifiers

	Format	Example
all	upnp:all	upnp:all
Unique Device Name (UDN)	upnp:UDN:*namespace:uniqueid*	upnp:UDN:IEEEMAC:0C009 9123456
Device Type	upnp:DevType:*devicetype*	upnp:DevType:vcr
Service Type	upnp:SrvType:*servicety*	upnp:SrvType:clock:1

pe:ver

SSDP specifies that SSDP announcements must be made for all SSDP searchable values. The SSDP announcements with "all" as the notification header value must carry the Root Device UDN as the USN header value. SSDP announcements for Device Types must carry the UDN of the Root Device concatenated with the Device Type URI as the USN header value. SSDP announcements for a Service Type will carry the UDN of the Root Device concatenated with the Service Type URI value as the USN header value. SSDP announcements of UDNs will repeat the UDN value as the USN header.

UPnP SSDP Announcements

Announcement	UPnP Notification Type	SSDP USN
	"all"	Root Device UDN
Unique Root Device	Root Device UDN	Root Device UDN
Unique non-Root Device	Non-Root Device UDN	Non-Root Device UDN
Device Type	Device Type Identifier	Root Device UDN + Device Type Identifier
Service Type	Service Type Identifier	Root Device UDN + Service Type Identifier

UPnP Bridges 120 (Figure 2) announce Bridged Devices 122-123 and associated UPnP Services using SSDP. The identifiers associated with the Bridged Devices are unique for the device, and they do not duplicate identifiers for Controlled Devices and UPnP Services directly available on the Bridge itself. This means that a Bridge that is also a Controlled Device must announce Bridged Devices and local Controlled Devices independently, with appropriate unique identifiers, Description Documents and associated URLs.

Description

The UPnP Description Document 226 (Figure 5) provides the information necessary to identify, describe, connect and control a UPnP Controlled Device 106-107 or Bridge 120 from a User Control Point 104-105.

The Description Document is an XML document. UPnP defines the use of HTTP and XML for the Description Document and wire protocols. UPnP adheres to the schema declaration rules of XML-Data and Y. Goland, "Flexible XML Processing Profile."

The top level XML elements are separated into three categories: per Device, per UPnP Service and shared.

Rehydrator

With reference now to Figure 9, all (UPnP) Controlled Devices 106-107 (Figure 1) or Bridges 120 (Figure 2) expose one or more UPnP Services 210-217 (Figure 5) that can be controlled remotely. Controlling such UPnP Services involves a message exchange between a User Control Point 104 and the device 106. This message exchange happens according to a

specific Service Control Protocol (SCP) 402, which specifies the content and sequence of the messages exchanged.

User Control Points 104 are not required to have any prior knowledge of the SCPs 402 required to control the UPnP Services on the various devices. Therefore, a Controlled Device or Bridge must be able to describe to a User Control Point the protocols required to control its UPnP Services, such that the User Control Point will be able to implement these protocols dynamically. This requires a standard way of declaring Service Control Protocols in a concise and unambiguous fashion. UPnP introduces a technique for declaring Service Control Protocols using a series of XML documents.

A Rehydrator 410 is a module that exposes a suitable API to applications and either invokes Commands on a UPnP Service or queries the state of that UPnP Service, or receives and responds to events. The primary job of the Rehydrator is to map between API calls and the Service Control Protocol sequence that invokes the Command.

As part of the Service Definition 406, a Service State Table 230 and Command Set 408 are defined. These things can be combined in a deterministic way defined by UPnP to produce a Service Control Protocol Definition (SCPD) 406, which includes a Service Control Declaration 404 and a Service Control Protocol 402. The SCPD 406 is a representation of the schema of a UPnP Service. It is possible to reconstruct the SST, Command Set and SCP from the SCPD.

The SCPD is directly embedded into the Description Document 226 of a Controlled Device. When the Description Document is uploaded into the User Control Point 104, the Rehydrator 410 can extract the SCPD from

it. At this point, the Rehydrator has enough information to issue UPnP Service specific SCPs 402.

General Operation of the Rehydrator

More generally with reference to Figure 10, the Rehydrator 410 operates as a universal adapter to provide a programmatic interface to any service-specific protocol of a remote computing device. The Rehydrator 410 simply obtains a data description or declaration of the methods, properties and events of the remote UPnP Service, as well as a definition of the protocol of network data messages through which the Rehydrator invokes the methods, queries or sets the properties, and receives event notifications. In UPnP, this data description takes the form of the Description Document 226, which contains a Contract 412. The Contract defines network data packets 413 (e.g., XML data), request/response patterns, and protocol (e.g., GENA, HTTP, SSDP) via which the packets are exchanged. This information is sufficient for the Rehydrator to exchange the appropriate network data packets to interact with the Controlled Device Service, including to invoke commands, query and set properties, and receive and respond to events, without download of any executable code to the User Control Point 104 device and with a zero installation or configuration experience.

The Description Document 226 also includes a declaration of the methods, properties and events for the UPnP Service. Based on this declaration, the Rehydrator produces a corresponding programmatic interface for use by applications at the User Control Point. The programmatic interface is an application programming interface that can be in the form of an object integration interface of an object-oriented

programming model, such as Microsoft COM, CORBA, Java classes, and scripting engine name extensions. In the example illustrated in Figure 10, the Rehydrator 410 exposes a COM object integration interface ("IClock" interface 414), with methods getTime() and setTime(), for a Controlled Device having a "Clock" UPnP Service with GetTime and SetTime commands. The Rehydrator 410 converts calls of an application program 416 to the IClock interface 414 into the network data messages specified in the Contract to invoke the corresponding commands of the Clock UPnP Service. The Rehydrator 410 likewise creates suitable further programmatic interfaces for other UPnP Services (e.g., UPnP Services 210-217 of Figure 5) based on the Description Document of their respective Controlled Devices.

Accordingly, the Rehydrator operates as a universal proxy object with data-driven conversion of programmatic interfaces to network data messages. Further, the Rehydrator produces the programmatic interface at the User Control Point based solely on an XML data description. This operation allows the Rehydrator to produce just-in-time transient interfaces to remote device UPnP Services without the complexity of code downloads and installation or configuration. Upon a later release of the interface by the application, the Rehydrator destroys the interface without need to de-install or clean up persistent configuration data in a registry or configuration file of the operating system or object execution run-time.

Rehydrator Implementation

Summary. With reference to Figure 11, a preferred implementation 440 of the Rehydrator 410 is as an internal Microsoft Windows component that routes service control requests from the UPnP API to devices.

Applications wishing to control a UPnP Service on a UPnP device obtain a Service object through the UPnP API and use the methods of this object to query the state variables of the UPnP Service and invoke its actions. Those methods use the Rehydrator API to turn the UPnP Service control requests into network messages that travel to the UPnP device. In this sense, the Rehydrator performs a mapping between API calls and network protocols.

Basic Functionality. The preferred implementation of the Rehydrator is able to translate a service control call to the UPnP API into the appropriate network messages defined by the Service Control Protocol.

Asynchronous Event Notification. The preferred implementation of the Rehydrator is able to notify UPnP API clients of any asynchronous events generated by the devices they are controlling. Event notification is done by means of the event interfaces defined below.

Error Reporting. For a variety of reasons, state variable queries and action invocations may fail. The preferred implementation of the Rehydrator is able to provide a way to communicate the success or failure status of such operations to the parties initiating them.

Rehydrator Implementation Design. As illustrated in Figure 11, the preferred implementation of the Rehydrator is used in two ways. First, the Device Finder 450 uses it to create Service objects 460. Then, these Service objects use it to carry out service control operations (querying state variables and invoking actions).

Creating Service Objects. When the Device Finder 450 creates a Device object, it invokes the Rehydrator 410 to create Service objects 460 for each of the service instances on that device. Each service instance supports a particular Service Control Protocol and the Rehydrator needs a

description of this protocol in order to create a properly hydrated Service object.

The Service Control Protocol is declared in two separate XML documents: the DCPD and the Contract. The Rehydrator needs the information in both documents. These two documents are passed to the Rehydrator as *IXMLDOMDocument* interface pointers in the *RehydratorCreateServiceObject()* API call.

```
HRESULT
RehydratorCreateServiceObject(
IN    IXMLDOMDocument *pDCPD,
      IN    IXMLDOMDocument *pContractDocument,
      OUT   IUPnPService    **pNewServiceObject);
```

This API returns a pointer to an *IUPnPService* interface on a newly created Service object. In addition to the creating the Service object, the Rehydrator sets up its internal data structures so that it can properly handle requests to control the UPnP Service. Specifically, it creates a list of the properties and actions exported by the UPnP Service. Since all service instances of the same service type export the same properties and the same actions, this information is kept only once for each service type and is indexed by Service Type Identifier.

The Rehydrator stores the information that is specific to a particular service instance as private data within the Service object itself. This includes the control URL and information about the control server 232 (such as the HTTP verbs it supports). The Service Type Identifier is the link between the Service object that represents one instance of a service type and the Rehydrator internal data structures that contain information

common to all instances of that service type. The Service Type Identifier is stored as a private data member in the Service object.

Querying Service Properties. Applications can query the values of service properties by invoking the *IUPnPService::GetProperty()* method on a Service object. Internally, this method makes a call to the *RehydratorQueryStateVariable()* function.

```
HRESULT
RehydratorQueryStateVariable(
     IN        LPCTSTR   lpcszVerb,
     IN        LPCTSTR   lpcszControlURL,
     IN        LPCTSTR   lpcszSTI,
     IN        LPCTSTR   lpcszVarName,
     OUT       VARIANT   *pValue);
```

The first two in parameters to this function supply the service instance specific information: the HTTP verb to use and the control URL to which the network messages will be targeted. The third parameter is the Service Type Identifier that will be used to locate the Service Control Protocol information in the Rehydrator's internal data structures. The fourth parameter is the name of the variable that is being queried (the Rehydrator will validate this against is internal list of state variables exported by the UPnP Service) and the final parameter is the address of a *VARIANT* structure in which the Rehydrator will place the variable's value.

This function will generate an HTTP request to the control server on the device. The body of this request will be an XML fragment containing a XOAP-encoded request for the variable's value. The following is an example of such a request (the exact header and payload format of this message is defined in the service contract):

```
M-POST /clockService HTTP/1.1
Host: spather-xeon:8586
Content-Type: text/xml
Man:
"http://www.microsoft.com/protocols/ext/XOAP";
ns=01
01-MethodName: queryStateVariable
01-MessageType: Call
Accept-Language: en-gb, en;q=0.8
Referer: http://myhouse/VCR1Presentation
Content-Length: 84
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01;
Windows NT 5.0)
Connection: Keep-Alive

<queryStateVariable>
     <variableName>currentTime</variableName>
</queryStateVariable>
```

The control server will respond to this message with another XML fragment: the XOAP-encoded method response. The following is an example of such a response:

```
HTTP/1.1 200 OK
Connection: Close
Cache-Control: private
Date: Mon Oct 11 12:13:38 PDT 1999
Expires: Mon Oct 11 12:13:38 PDT 1999
Content-Type: text/xml
Content-Length: 62
Man:
"http://www.microsoft.com/protocols/ext/XOAP";
ns=01
01-MessageType: CallResponse

<queryStateVariableResponse>
     <_return>12:13:28</_return>
</queryStateVariableResponse>
```

The rehydrator will extract the return value from this XML fragment, place it in the *VARIANT* structure whose address was passed as the last parameter to *RehydratorGetServiceProperty()* and then return.

Invoking UPnP Service Actions. The process of invoking a UPnP Service action is very similar to querying a state variable. An application calls *IUPnPService::InvokeAction()* on a Service object, passing it the name of an action to invoke, and an array of arguments to the action. Internally, *IUPnPService::InvokeAction()* calls *RehydratorInvokeServiceAction()*, declared as shown below.

```
HRESULT
RehydratorInvokeServiceAction(
    IN          LPCTSTR    lpcszVerb,
    IN          LPCTSTR    lpcszControlURL,
    IN          LPCTSTR    lpcszSTI,
    IN          LPCTSTR    lpcszActionName,
    IN          SAFEARRAY  saActionArgs,
    OUT   LONG             *pStatus);
```

As was the case for querying state variables, the service instance specific information is passed in the first two parameters, followed by the Service Type Identifier in the third. The action name and an array of arguments are passed as the next two parameters, and the final parameter is the address of a variable in which to store the status of the operation.

RehydratorInvokeServiceAction() will send an HTTP request to the control server identified by the second parameter. As before, the body of this message will be an XML fragment containing a XOAP-encoded method call. An example HTTP request to invoke an action is shown below.

```
M-POST /clockService HTTP/1.1
Host: spather-xeon:8586
Content-Type: text/xml
Man:
"http://www.microsoft.com/protocols/ext/XOAP";
ns=01
01-MethodName: invokeAction
01-MessageType: Call
Accept-Language: en-gb, en;q=0.8
Referer: http://myhouse/VCR1Presentation
Content-Length: 119
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01;
Windows NT 5.0)
Connection: Keep-Alive

<SerializedStream main="invokeAction">
     <invokeAction id="invokeAction">

     <actionName>setCurrentTime</actionName>
          <actionArg>15:41:29</actionArg>
     </invokeAction>
</SerializedStream>
```

The encoding of the body of this message is again specified in the service contract. The Rehydrator will wait for the HTTP response to this request, which would look something like the example below.

```
HTTP/1.1 200 OK
Connection: Close
Cache-Control: private
Date: Mon Oct 11 15:22:38 PDT 1999
Expires: Mon Oct 11 15:22:38 PDT 1999
Content-Type: text/xml
Content-Length: 50
Man:
"http://www.microsoft.com/protocols/ext/XOAP";
ns=01
01-MessageType: CallResponse

<invokeActionResponse>
     <_return>0</_return>
</invokeActionResponse>
```

After receiving a response such as this, the Rehydrator will extract the return value, place it in the out parameter it was passed, and then return.

Figures 30 through 40 are program listings defining various interfaces used in the preferred implementation of the Rehydrator, including an IUPNPDevice Interface, an IUPNPPropertyBag Interface, an IUPNPService Interface, an IUPNPDevices Interface, and an IUPNPServices Interface.

Description Document

With reference to Figure 15, User Control Points 104 can retrieve a Description Document 226 by issuing an HTTP GET on a Description URL. This URL is returned in the location header of either an SSDP announcement or an SSDP query response.

The HTTP GET must include an accept-language header that is used to request the preferred language of the response. If the requested language is not supported, a Description Document in the default language supported by the Controlled Device or Bridge may be returned.

An HTTP GET is used to retrieve sub elements of a Description Document that are expressed as URLs.

URL Handling

URLs embedded in Description Documents 226 take one of 3 forms: a fully qualified URL or a relative URL.

Fully qualified URLs take the form:

```
http://devicename/pathname
```

The devicename part of the URL is a Hostname or IP address and the pathname is a filesystem path or equivalent. A fully qualified URL is used "as is" to establish an HTTP connection to a device.

A relative URL does not contain the ":" character and is of the form:

```
pathname
/pathname
```

Relative URLS are a compact representation of the location of a resource relative to an absolute base URL. All relative URLs in a Description Document are appended to the value of the Description Document element <URLbase> to form fully qualified URLs.

Binary Data

Some elements of a Description Document are binary. XML does not directly support the embedding of binary data. In order to include binary data directly in a Description Document, one must convert the data to text using the Base 64 encoding scheme. This tends to increase the size of the data by 25% on the average. Much of this overhead can be eliminated if the binary data is passed by reference instead of by value. To reference binary data, a URL to the data is provided in a Description Document. The binary data can be retrieved by doing a HTTP GET with that URL.

As an example, consider the <image> element in the following Description Document:

```
<iconList>
    <icon>
        <size>16</size>
        <imageType>PNG</imageType>
        <color>1</color>
        <depth>8</depth>
        <image>
"http://device.local/iconpath/icon.png"/>
```

```
    </icon>
</iconList>
```

The icon would be retrieved with an HTTP GET of the following format:

```
GET iconpath/icon.png HTTP 1.1
Host: device.local
```

The HTTP response would look like:

```
HTTP/1.1 200 OK
Content-Type: image/png
Content-length: ###
<binary color icon data in the PNG format>
```

Description Document Layout

The basic layout of the Description Document 226 is shown in Figure 16.

The following table lists Description Document elements that are sub-elements to the root element.

Root	The XML root element of a UPnP Description Document.
specVersionMajor	The major version of the UPnP Architectural Reference that this Description Document was created against. This value must be 1.
specVersionMajor	The minor version of the UPnP Architectural Reference that this Description Document was created against. This value must be 0.
URLBase	An optional element used to construct fully qualified URLs.

	Relative URLS are appended to the value of <URLBase> to create fully qualified URLs. If this element is present, it must agree with the HTTP Base header.
manufacturer	A required element that contains a textual manufacturer name.
manufacturer URL	An optional element containing a URL that points to the Web page of the manufacturer.
modelName	A required element containing a textual product name.
modelDescrip tion	A required element containing a textual product description.
modelNumber	An optional element containing a textual product model number.
modelURL	An optional element containing a URL that points to the Web page of the product.
UPC	An optional element containing the product Universal Product Code (UPC).
serialNumber	An optional element containing a textual item serial number.

The Description Document elements listed in the following table are associated with devices.

rootDevice	A required sub element of the root. This element is a container for one or more service elements and the elements that describe the rootDevice.

device	An optional sub element of the root or another device element. This element contains the same kinds of elements as a rootDevice element.
UDN	A required sub element of every rootDevice or device element containing the Unique Device Name.
friendlyName	A required sub element of every rootDevice or device element containing a textual friendly name. This element can be updated remotely.
deviceType	A required sub element of every rootDevice or device element containing a standardized Device Type Identifier.
presentation URL	An optional sub element of a rootDevice or device element containing a Presentation URL.
iconList	A required sub element of every rootDevice or device element. This element is a container for one or more icon elements. UPnP requires a base set of six icons that must exist in the iconList. All devices must support PNG icon image formats of three sizes, 16 by 16, 32 by 32 and 48 by 48 pixels in both color and black and white at 8 bit depth. Additional formats and sizes, including JPEG, GIF, BMP, ICON and VML, may be supported by adding them to the list.
icon	A required sub element of every iconList element. This element is a container for the elements that define an icon.
size	A required sub element of every icon element. There must be icon elements with associated size elements with the values

	16, 32 and 48. Other icons may specify other sizes.
color	A required sub element of every icon element with value 0 or 1. Each icon of size 16, 32 or 48 must exist in color and black and white.
depth	A required sub element of every icon element. All required icons must exist with a value of 8.
imageType	A required sub element of every icon element that identifies the format of the binary icon: png, jpeg, vml, gif, bmp, or ico.
image	A required sub element of every icon element that references a binary icon.

The following elements of the Description Document are associated with UPnP Services.

service	An optional sub element of the rootDevice or another device element. This element is a container for the Service Definition.
serviceType	A required sub element of every service element containing a standardized Service Type Identifier.
controlURL	A required sub element of every service containing a Control URL.
eventSubURL	A required sub element of every service containing an Event Subscription URL.
SCPD	A required sub element of every service. The SCPD is a container for the standardized Service Control Protocol

Declaration associated the Service.

Figure 17 shows an exemplary icon list in a Description Document 226.

Service Control Protocol and SCP Declaration

As part of the Service Definition 406 shown in Figure 9, a Service State Table 230 and Command Set 408 are defined. The SCPD 406 is a representation of the schema of a UPnP Service. It is possible to reconstruct the SST 230, Command Set 408 and SCP 402 from the SCPD deterministically.

The declaration of such a protocol must specify the list of Variables that can be queried, the set of Commands that can be invoked, as well as the wire protocol (the content and sequence of network messages) required to carry out these operations. SCPD is specified in two XML documents. The first or Service Control Definition document 404, written in a language called Service Control Protocol Declaration Language (SCPDL), declares the list of state Variables and Commands associated with the Service Type to be controlled by the protocol. The second or Service Control Protocol document 402 declares the wire protocol that will be used to query the values of the state variables and invoke the actions associated with the UPnP Service.

Declaring the Service State Table and Command Set

A SCPDL document 404 is used to specify the list of state Variables that a SCP can query and the set of Commands that it can invoke. SCPDL is an XML schema, a set of rules for writing XML documents (Service Control Protocol Declarations).

Figure 18 shows an exemplary SCPDL document. This XML document consists of a root *<scpd>* element containing two sub-elements, *<serviceStateTable>* and *<actionList>*. Within the *<serviceStateTable>* element is a *<stateVariable>* element for each state variable associated with the UPnP Service. The UPnP Service in this example is a TV tuner with has only one state variable, *currentChannel.* The elements within the *<stateVariable>* element specify the name, data type and allowed values for the state variable. Had the UPnP Service more state variables, they would be represented by additional *<stateVariable>* elements within the *<deviceStateTable>* element.

The *<actionList>* element contains an *<action>* element for every action associated with the UPnP Service. The elements within an *<action>* element specify the name of the action and any arguments the action may take. In this case, the UPnP Service supports two actions that do not take arguments, *ChannelUp* and *ChannelDown*, and another, *SetChannel*, that takes a new channel number as an argument. The *<argument>* element and the elements nested within it define the argument. The *<relatedStateVariable>* element within *<argument>* specifies the name of one of the state variables to which the argument is related. In the UPnP Device Model, all arguments to actions must correspond directly to some state variable.

Figures 19 and 20 show an XML schema for the SCPDL.

Basic UPnP Eventing Architecture

With reference to Figure 21, the UPnP architecture 200 (Figure 5) requires that clients of the UPnP API be enabled to receive notifications reliably from UPnP services 210-217 as their states change. Since state

changes are relatively common, the eventing subsystem's efficiency and performance is a major consideration in this design. Figure 21 and the following discussion describe the Basic UPnP Eventing Architecture 600, which encompasses both the controlled device (CD) 106 and user control point (UCP) 104 sides of the eventing UPnP Service. It also includes the support APIs for both a low-level UPnP Service interaction and a higher level COM-based wrapper of those APIs. The latter enables automation controllers like Visual Basic and JScript 602 to receive event notifications.

What is an event?

Property change events are defined as any change in the value of a row of the Device State Table (DST) 230 (Figure 5) for a UPnP Service 210-217. This change will be reflected as a property change notification. For example, if a "VCR" device has a "VCR Transport" UPnP Service, one row in that UPnP Service's DST may be *TapeState* and the value could be *TapePresent*. If the tape is ejected, the new value would be *TapeAbsent*. This state change would be reflected as a notification sent to all subscribers.

What is a notification?

A UPnP event notification is an XML message sent over HTTP/TCP to each and every subscriber to a particular UPnP service. The content of the XML is defined below. The important contents of this message are the unique identifier for the subscription, the property name, new value, and property type.

Notification Processing

In UPnP, the listener to Notifications is the SSDP service itself. SSDP already listens on another multicast address for "alive" and "byebye" messages sent by UPnP devices. The same listener will listen on a TCP port for notifications sent. All subscriptions sent from that UCP contain the same callback URL and so all notifications will be directed to that URL. When a notification arrives the SSDP service will examine the NT header of the message and determine if it is an event notification. If so, the message is parsed further to determine if it should be forwarded on to subscribers (which must exist). GENA defines the format of the HTTP message, what headers can be used, and what they can be used for.

GENA

GENA is the protocol of communication that, in a preferred embodiment, UPnP devices use to send event notifications. Therefore, UPnP devices that wish to notify UCPs of state changes are recommended to use GENA. Notification subscribers will never be required to interact with a UPnP device directly and so they are not required to use GENA. The eventing API will encapsulate this complexity. Other appropriate event transport protocols may be used, such as publish/subscribe systems.

Receiving Notifications

Applications written in C (C Application 604) will be able to utilize the SSDP C API 610 to receive callbacks when notifications are processed by the SSDP service. This is analogous to SSDP clients registering for notifications that services have become available. When a UCP registers for a notification, it passes as a parameter the URL of the UPnP Service for which it is interested in receiving notifications. This URL is obtained from

the description document for that UPnP Service. (When a UPnP Service is registered on a UPnP device, it uses this same URL to listen for subscription requests).

When a notification message is received by the SSDP service listener, the SID header is checked against the list of subscribers it maintains. If a subscriber is found, the callback function for that subscriber is invoked, with one of the parameters being the contents of the notification message. The notification client that implements the callback function can process this message in any appropriate way.

Notifications in the UPnP API

The UPnP API 410 is a consumer of the basic C interface provided by the SSDP C API 610 component. In order to integrate seamlessly, the registration of notifications is handled by the Service Object 612 inside the UPnP Object Model. Service objects will register for notifications when they are created. This ensures that the DST is maintained by the UPnP API and is kept up to date. They will implement the callback function required by the registration function. If this callback function is invoked, it will pass on that notification to UCPs. The UCPs can be written in C, C++, VB, or script code, so the mechanism for passing on notifications can be different.

Script Support

A feature of the illustrated eventing system is that it supports script languages such as VBScript and JavaScript 602. For VBScript, this is made possible by providing a property on the Service object that, when set, contains the IDispatch pointer for a VBScript function or subroutine that will be the event handler. When the Service object's notification callback is invoked, it checks to see if this IDispatch pointer was set, and if

so, it calls IDispatch::Invoke on DISPID 0 of that interface to call the VBScript subroutine. An equivalent mechanism is implemented for JScript.

Eventing Subsystem Terminology

UCP – User control point. Any piece of software that searches for devices and controls them.

CD – controlled device. A hardware or software device that announces its availability thru SSDP and allows control by UCPs.

Subscriber – A UCP who wishes to be notified of event changes.

Notifying Resource (or simply "Resource") – For the purposes of this document, this will *always* be a UPnP Service contained within a UPnP CD 106.

Event Source – a UPnP Service that provides events. UPnP services are event sources. All notifying resources are event sources and vice versa.

Event – message generated when a change in a resource's state occurs.

Property – a single entry in the UPnP Service's state table whose DefaultValue can change. Properties and events always have a one to one correspondence.

Subscribing To Resources

Integrating With The UPnP API

The UPnP API 410 exposes several interfaces with which a consumer can find and enumerate devices, control UPnP Services, and get properties on devices and UPnP Services. To allow the integration of events into this model, we add a new property to the IUPnPService

interface called *EventHandler*. When this property is set, it tells the Service object 612 that its client is interested in receiving notifications for that UPnP Service. The SSDP API RegisterNotification() API is called when the Service object is created so that it can maintain a local copy of the DST for that UPnP Service. The Service object knows the URL of the UPnP Service and therefore it can provide this as a parameter to RegisterNotification(). RegisterNotification() is also provided a callback function which is a static member of the Service object class. This function will be invoked for each and every notification sent by that particular UPnP service.

The Notification Callback

The Service object 612 includes a static member function called *EventNotifyCallback()* which is invoked for each notification sent by the UPnP service. The callback is passed the entire HTTP message contents in a structure which is a parameter to the function. The prototype looks like this:

```
static VOID
CUPnPService::EventNotifyCallback(SSDP_CALLBACK_
TYPE ssdpType,
SSDP_MESSAGE *pssdpMsg,
                          LPVOID pcontext);
```

The *ssdpType* parameter should always be SSDP_PROPCHANGE. The *pssdpMsg* parameter contains the relevant information about the event. The key piece of information is the body of the XML message. The body contains information about what property changed, what its new value is and what type it is, among other information. The *pContext* parameter will always be the *this* pointer of the Service object. This allows the code to call a method to fire the event to the UCP. The callback will

parse the XML body using the XML DOM services. Property changes are iterated and the local DST is updated to reflect these changes. After this processing is done, an event notification may be fired for each property that was changed to the owner of the subscription if one exists. Depending on what environment the owner is written in (C++ or script, etc...), a different mechanism for firing the event may be employed.

A special case for this process is the very first notification received after a subscription is established. This notification contains the entire set of properties and their values and is used to locally sync up the DST. Events will not be fired to clients of the UPnP API in this case.

Firing Notifications

When the EventNotifyCallback() function is called, the local copy of the DST for the UPnP Service is updated. After this, an event needs to be fired if a subscriber exists. A subscriber exists if the put_EventHandler() method was called, either from VBScript, C++ code, or another source. To abstract away this complexity, a new interface called IUPnPEvents is needed.

This interface currently has one method called NotifyEvent() which takes several parameters. When put_EventHandler() function is called, its argument is an IUnknown. This pointer is QueryInterface'd() for IDispatch first, and if it succeeds, then IDispatch::Invoke() is called with DISPID 0 to invoke the default method. This allows VBScript 602 to be called. If that fails, however, it is Queried for IUPnPEvents, and if that succeeds, the NotifyEvent() method is called with the same parameters as for Invoke(). The handles C++ UCPs effectively.

Subscribing with C++

To subscribe to a UPnP service from C++, a UCP instantiates a UPnP service object, issues QueryInterface() to it for IUPnPEvents, and calls the IUPnPEvents::SetEventCallback() function. This function takes 2 parameters, a callback function pointer and a context pointer.

Subscribing With VBScript

To subscribe to a UPnP service's events, all that needs to be done by a script 602 is to create a function or subroutine as a handler function and set the pointer of that function to the *EventHandler* property of the Service object. Now, anytime an event is fired, this VBScript function or subroutine will be called. In VBScript, this is written as the following:

```
Dim UPnPAPI
Set UPnPAPI = CreateObject("UPnPAPI.1")

Devices = UPnPAPI.FindDevices(...)
For each device in Devices
     For each service In devices.services
          If service.dcpi = "clock.v1"
               Service.EventHandler =
GetRef("clock_PropertyChanged")
          End if
     Next service
Next device

Sub clock_PropertyChanged(prop, value)
     MsgBox "The time has changed. It is now " &
value & "."
End Sub
```

In this example, the script enumerates all devices, looking for any device that supports the "Clock" interface. When it finds a device that supports that interface, it enumerates that device's UPnP Services looking for the one that has the "clock.v1" interface. Once it finds that UPnP

Service, it sets that service's *EventHandler* property to the VBScript subroutine called "clock_PropertyChanged". This name is arbitrary.

Sending and Receiving Notifications

GENA Client API

GENA clients are actually UPnP *services*. A GENA client creates a new event source when it is initialized. The GENA client API 620 facilitates this. It also provides a way for GENA clients to send their notification messages. It is also important to note that the HTTP server that lives on the UPnP device is also a client of this API. The GENA client API consists of the following functions:

RegisterUpnpEventSource()

The RegisterUpnpEventSource() API gives a GENA client the ability to register itself as an event source. The prototype is as follows:

```
BOOL RegisterUpnpEventSource(
LPTSTR szRequestUri,
DWORD cProps,
UPNP_PROPERTY *rgProps
);
```

Parameters: *szRequestUri* [in] an arbitrary Request-Uri that SUBSCRIBE requests will be sent to. When a SUBSCRIBE request arrives at the given URI, it is acknowledged and the subscriber is added to the list of notification recipients. Note that this URI should match the URI provided in the description for this service. *CProps* [in] the number of properties that this event source provides. *RgProps* [in] Array of UPNP_PROPERTY structures which contain information about each property. The property information is derived from the DST for the event source.

Return Value: The function returns a TRUE if successful. If the given URL has already been registered as an event source, the return value is FALSE and GetLastError() returns ERROR_ALREADY_EXISTS.

Notes: The initial state of the event source needs to be given to the API so that it can effectively maintain the up-to-date state of the event source.

DeRegisterUpnpEventSource()

The DeRegisterUpnpEventSource() API gives a GENA client the ability to deregister itself as an event source. The prototype is as follows:

```
VOID DeRegisterUpnpEventSource(
     LPCTSTR szRequestUri
);
```

Parameters: *szRequestUri* [in] an arbitrary Request-Uri that SUBSCRIBE requests will be sent to. When a SUBSCRIBE request arrives at the given URI, it is acknowledged and the subscriber is added to the list of notification recipients. Note that this URI should match the URI provided in the description for this UPnP Service.

UPNP_PROPERTY

```
typedef struct _UPNP_PROPERTY {
     LPTSTR szName;
     LPTSTR szValue;
     LPTSTR szType;
} UPNP_PROPERTY;
```

Where *szName* is the name of the property, *szValue* is the current value of property, and *szType* is the type of property (string, integer, etc...).

SubmitUpnpPropertyEvent()

The SubmitUpnpPropertyEvent() API allows the GENA client to submit a UPnP property change event to be sent to subscribers as a notification. The prototype is as follows:

```
BOOL SubmitUpnpPropertyEvent(
LPCTSTR szRequestUri,
DWORD dwFlags,
     DWORD cProps,
      UPNP_PROPERTY *rgProps
);
```

Parameters: *"szRequestUri* [in]" identifies the event source to which this event belongs. This is the same Request-Uri passed to RegisterUpnpEventSource(). *"DwFlags* [in]" is unused. *"CProps* [in]" is the number of events that are being submitted. "R*gProps* [in]" is an array of UPNP_PROPERTY structures which contain information about each event.

Return Value: If the function fails, the return value is FALSE. The get extended error information, call the *GetLastError()* function.

Notes: When a series of properties is submitted for event notification, the local version of the property state for the given event source is updated with the list of properties passed in. SubmitUpnpPropertyEvent() calls SubmitEvent() after it has generated an XML body.

SubmitEvent()

The SubmitEvent() API allows the GENA client to submit an unstructured event to be sent to subscribers as a notification. The prototype is as follows:

```
BOOL SubmitEvent(
LPCTSTR szRequestUri,
DWORD dwFlags,
```

```
        LPCTSTR szHeaders,
        LPCTSTR szEventBody
);
```

Parameters: *SzRequestUri* [in] identifies the event source to which this event belongs. This is the same Request-Uri passed to RegisterUpnpEventSource(). *DwFlags* [in] Unused. *SzHeaders* [in] null-terminated text string containing the headers for the event, each separated by CRLF. SzEventBody [in] null-terminated text string containing the body of the event message

Return Value: If the function fails, the return value is FALSE. The get extended error information, call the ***GetLastError()*** function.

Notes: If no subscribers exist, the function does nothing. If one or more subscribers exist, a message is sent to each subscriber. SubmitEvent() will always send to **all** subscribers.

UPnP Controlled Device Event Architecture

In UPnP, every UPnP service 210-211 that supports property change event notifications is to be a GENA client. Therefore, when the UPnP Service is initialized, it must register itself as a GENA event source. It will do this with the RegisterUpnpEventSource() API. This returns a handle which can be used in subsequent APIs.

RegisterUpnpEventSource() takes a URL and an array of properties as parameters. Inside the API, an entry in an array of structures is initialized and the index is returned as the handle. The structure contains the source URL as one of the members. A second member of the structure, an array of destination URLs, is left uninitialized. This is filled in each time as subscriber is added for that event source. Another member of the structure is the list of properties that this event source provides. This is

effectively a cached copy of the DST for the event source. As events are submitted, the local properties are updated.

When SubmitUpnpPropertyEvent() is called, each property submitted replaces the corresponding property already maintained by the API. If no subscribers exist, the request to submit an event is ignored. If one or more subscribers exist, their callback URLs are looked up in the list of subscribers for the given event source and a NOTIFY message is constructed and sent to each URL, one at a time, in order of subscription.

If an event is submitted and no response is received (or a CD-side error occurs), the CD continues to attempt to send to the UCP. If the subscription timeout expires, then the subscription is removed. If the UCP becomes available again, it will re-subscribe because it will notice the sequence numbers are not contiguous.

When an HTTP server 626 receives a SUBSCRIBE message, it passes it along to a function which parses the message for the necessary information. The Request-URI identifies the UPnP Service that is to be subscribed to. The callback URL is obtained from the "Callback" header. Since the Callback header can contain multiple URLs, it picks the first "http://" URL it finds. It then adds the subscriber to the list of subscribers for this event source. A unique subscription identifier is constructed which it will send back to the subscriber in the HTTP response to the SUBSCRIBE request.

If no event source matches the Request-URI from the subscription message, the HTTP server should return "404 Not Found".

When a subscription is added, the local copy of the DST is sent as a NOTIFY message. This special NOTIFY message contains sequence

number 0 which informs the UCP that this is an initial state population event and not a notification where every event has changed.

When a CD receives an UNSUBSCRIBE message, it checks the "SID" header to obtain the subscription identifier. It looks up the subscriber ID in the list of subscribers for that event source and removes the destination URL entry associated with it.

GENA Server API

GENA servers 630 are generally going to be UPnP UCPs. A GENA server is anything that receives and processes NOTIFY messages to handle notifications from resources and sends SUBSCRIBE and UNSUBSCRIBE messages to receive notifications from resources. These APIs leverage the already existing SSDP APIs. The following are the changes to the APIs:

RegisterNotification()

The RegisterNotification() allows a UPnP UCP to request notification when an event occurs for a given UPnP service. The prototype is as follows:

```
HANDLE RegisterNotification(
NOTIFY_TYPE    nt,              // SSDP_ALIVE |
SSDP_PROPCHANGE | ??
LPTSTR szResourceType,          // based on
NOTIFY_TYPE, unused if
// SSDP_PROPCHANGE is used.
LPTSTR szEventUrl,
ServiceCallbackFunc fnCallback,
void *pContext
);
```

Parameters: *Nt* [in] An enumeration that determines the type of notification requested. The values are: SSDP_ALIVE – a UPnP Service has

become available, and SSDP_PROPCHANGE – a property has changed on the UPnP Service. SzResourceType [in] A null-terminated string specifying the resource type desired. For SSDP_ALIVE, this is the service type, for SSDP_PROPCHANGE this is unused. *SzEventUrl* [in] A null-terminated string specifying the URL that a subscription request should be sent to. *FnCallback* [in] A pointer to a function that will be called each time a notification is received. The function pointer is defined in the SSDP spec. *PContext* [in] This parameter is included as a parameter when invoking the client-supplied callback function.

Return Value: If the function succeeds, the return value is a handle used in a subsequent call to the DeregisterEventNotification() function. If the function fails, the return value is INVALID_HANDLE_VALUE error code. To get extended error information, call GetLastError.

ServiceCallbackFunc

```
typedef enum _SSDP_CALLBACK_TYPE {
    SSDP_FOUND = 0,
    SSDP_ALIVE = 1,
    SSDP_BYEBYE = 2,
    SSDP_DONE = 3,
    SSDP_PROPCHANGE = 4,
} SSDP_CALLBACK_TYPE, * PSSDP_CALLBACK_TYPE;
```

UPnP UCP Architecture

When a UPnP UCP wishes to subscribe to notifications for a particular UPnP service, it calls the RegisterNotification() API. It passes to this API a notification type that identifies the type of notification being requested, a URL to which a subscription should be sent, and a callback function and context for use when the notification is received.

RegisterNotification() will compose a SUBSCRIBE message, using the data passed in, and send that to the URL specified by the caller. The Callback header of the SUBSCRIBE message will be composed on the fly, as an arbitrary URL for notifications to be sent to for this subscription. This callback URL will likely be a constant since the server API will always know how to handle requests sent to this URL. It will then send the SUBSCRIBE message and await a response.

RegisterNotification() in the SSDP API does not currently send HTTP requests, but it can be modified to do so. It also needs to await a response which it will also be modified to do so.

When the response is received, the Subscription-ID header contains a SID which is associated with the callback function specified by the caller.

Immediately after the response is received, the UCP should expect an initial NOTIFY message that contains the complete set of properties maintained by the CD. This becomes the local cached DST on the UCP side. From this point on, all modifications to the table are made via NOTIFY messages. This initial NOTIFY message will have sequence number 0 that indicates it is an initial property set and not an update. The UCP can use this information in any way it sees fit. This ensures the UCP's state table is always in sync with the one on the CD.

When a message is received by the HTTP server on the UPnP UCP, it is passed to a function which determines the method name and Request-URI. If this is a NOTIFY message, the headers are parsed and packaged up into a structure. The callback function that was specified to RegisterNotification() is called with that structure as one of the parameters. UCPs who implement the callback function can find the headers and body

of the NOTIFY message and do additional processing based on the notification type.

This all requires that the SSDP HTTP server listen on a TCP socket in addition to the UDP multicast port it already listens to. However, once a NOTIFY message is received, it is processed in the same way regardless of from which connection it originated.

Handling Failures

The following are subscription/notification failures that can occur and their solutions:

Leaked Subscriptions

To protect against subscriptions that exist on the controlled device, but no longer on the UCP, we institute the timeout feature of GENA subscriptions. The scenario is this: A UCP subscribes to a CD, then the UCP reboots. Meanwhile, the CD is still trying to send notifications to that UCP. If the UCP never comes back, the subscription would be leaked because the UCP never told the CD that it was going away. So to correct this, each subscription request includes an arbitrary timeout value which indicates to the CD that the UCP will be re-subscribing every *n* seconds indicated in the timeout header of the subscription request. If the timeout expires on the CD, the subscription is removed. The UCP is required to re-subscribe before the timeout period has elapsed. If it fails to do so, the subscription will be terminated by the CD.

Some time before the timeout expires on the UCP, a re-subscribe message should be sent. The re-subscribe message is similar to the subscribe message, but it does not contain an NT or Callback header. If the UCP is unable to re-subscribe within the timeout period, the

subscription will be terminated by the CD. If the UCP sends a re-subscribe after the CD has terminated the subscription, the CD will return "412 Precondition Failed".

Reboot of a Controlled Device

If a controlled device reboots, information about all of its subscribers would be lost. To prevent this, the subscriber information will be persisted across reboots of the device. Because the subscription info contains a timeout member, the absolute expiration time will be used when the subscription information is persisted. That way, when the device comes back up, it can check the timeout for each subscriber and if that time has passed, the subscription will be removed.

Network Error Sending Event Notifications

If a controlled device receives an error sending an event notification to a subscriber, it will **NOT** cease to send notifications. It will continue to send notifications and receive errors until the subscription expires. The problem for the UCP is that it will have missed a number of event notifications and so its state table will be out of sync. To correct this, each event notification message will contain a 32-bit sequence number that starts at 0 and increments for each message sent to a subscriber. If a subscriber receives a notification with a sequence number that is not exactly one more than the previous notification, it will know that it has lost events and will ignore all future notifications until it receives one with sequence number 0 again. Events with sequence number 0 indicate that the event is an "initial state" event.

Once it realizes that is has lost one or more events, the UCP will send an UNSUBSCRIBE message, followed by a SUBSCRIBE message. This

is not the same as a re-subscription because re-subscriptions do not cause the CD to start the sequence over at 0. In this case, the active unsubscribe/subscribe will cause the CD to restart the sequence at 0 and send the entire state table with the first notification message.

The SUBSCRIBE Message

When a UPnP UCP wishes to subscribe to event notifications for a UPnP service 210-211, it will form a SUBSCRIBE message of the following format:

```
SUBSCRIBE service1 HTTP/1.1
Host: vcr.local:200
NT: upnp:event
Callback: <http://remote1.local:923/upnp>
Timeout: Second-600
```

The response is as follows::

```
HTTP/1.1 200 O.K.
SID: uuid:kj9d4fae-7dec-11d0-a765-00a0c91e6bf6
Timeout: Second-600
```

This example of a GENA SUBSCRIBE request and response demonstrates a subscription to event notifications for "service1." The host is "vcr.local." All notifications for this UPnP Service will be sent to the callback URL http://remote1.local:923/upnp. In the response, the "Subscription-ID" header provides the subscriber with an identifier to use when it wants to unsubscribe to this resource. The "Timeout" header indicates that the subscriber will send a re-subscription request before 10 minutes have elapsed. If the device does not receive this request within that period of time, it will remove the subscription.

The Re-SUBSCRIBE Message

When a UPnP UCP wishes to re-subscribe to event notifications for a UPnP service, it will form a SUBSCRIBE message of the following format:

```
SUBSCRIBE service1 HTTP/1.1
Host: vcr.local:200
SID: uuid:kj9d4fae-7dec-11d0-a765-00a0c91e6bf6
Timeout: Second-600
```

The response would be as follows::

```
HTTP/1.1 200 O.K.
SID: uuid:kj9d4fae-7dec-11d0-a765-00a0c91e6bf6
Timeout: Second-600
```

Note that the NT and Callback headers are absent, but the SID header exists. This tells the CD 106 which subscription is being renewed and restarts the timeout. When the CD receives this message, it will persist the subscriptions to disk (or other persistent data storage medium), updating the absolute timeout based on the current time and a new timeout sent by the UCP (if it was different).

The NOTIFY Message

When a resource wishes to send an event notification, it will form a NOTIFY message of the following format:

```
NOTIFY upnp HTTP/1.1
Host: remote1.local:923
NT: upnp:event
NTS: upnp:propertychanged
SID: uuid:kj9d4fae-7dec-11d0-a765-00a0c91e6bf6
Seq: 123
Content-Length: xxx
Content-Type: text/xml

<event XML schema>
```

The response is as follows::

```
HTTP/1.1 200 O.K.
```

This example of a GENA NOTIFY request and response demonstrates that a "upnp:propertychanged" event is being sent to http://remote1.local:923/upnp. The USN header identifies "vcr.service1" as the event source. The XML contains the property name, value, and type. The "Seq" header indicates the sequence number of the notification. Sequence number 0 indicates the initial state update for the subscriber.

Property Change Event XML Schema

A UPnP property change event will be of the following form:

```
<U:propertyset xmlns:U="upnp">
   <U:propcount>2</U:propcount>
<U:property>
<U:foo>
     <U:type>string</U:type>
     goodbye
</U:foo>
</U:property>
<U:property>
<U:bar>
     <U:type>integer</U:type>
     27
</U:bar>
</U:property>
</U:propertyset>
```

Here, a property named "foo" is of type "string" and has a value of "goodbye" and a property named "bar" has a type of "integer" and has a value of 27. The XML will be contains a list of multiple properties that have changed, along with a count to make it easy to determine this.

The UNSUBSCRIBE Message

When a UPnP UCP wishes to unsubscribe to event notifications for a UPnP service, it will form an UNSUBSCRIBE message of the following format:

```
UNSUBSCRIBE service1 HTTP/1.1
Host: vcr.local:200
SID: uuid:kj9d4fae-7dec-11d0-a765-00a0c91e6bf6
```

The response would be as follows::

```
HTTP/1.1 200 O.k.
```

This example of a GENA UNSUBSCRIBE request and response demonstrates that the UCP is no longer interested in receiving event notifications from http://vcr.local/service1:200.

Step By Step: UCP to CD & Back

This section will take a step by step approach to what happens on both sides (UCP & CD) of an event notification. The description starts at the initialization of a UPnP device. Figure 22 illustrates the subscription, notification, and unsubscription process.

1. A UPnP device called "vcr" initializes.
    a. It sets itself up to be an HTTP server by doing the following:
        i. It binds to a TCP socket using its IP address and an arbitrary port number. This address/port pair will be referenced by all incoming URL requests.
        ii. It listens for incoming connection requests on that socket and sets itself up to accept any incoming connections.
    b. It sets itself up to be an HTTP client by doing the following:

i. Calls InternetOpen() to get a handle to the internet session

c. For each UPnP Service it exposes, it does the following:

i. It calls the SSDP API RegisterUpnpEventSource() to let the SSDP server know that it will be accepting subscriptions and sending event notifications. At this point, it has no subscribers. Note that this is called before the UPnP Service has announced itself so that it can be ready to accept subscriptions immediately. RegisterUpnpEventSource() sends no network traffic on the wire. It is a local initialization only. RegisterUpnpEventSource() does the following:

1. Adds a structure to the list of event sources containing the following:

a. A URL to which subscribers will send subscription requests

b. A list of destination URLs. A notification message will be sent to each destination URL.

c. The state table for the event source. This structure contains the property name, value, and type for each property supported by the UPnP Service.

ii. It calls the SSDP API RegisterService() to let the world know that it has become available. RegisterService() will send out an SSDP "alive" message on the multicast channel that will be heard by any device running the SSDP service.

d. It starts sending events immediately, even without subscribers. Each event submission updates the local state table. This submission needs to be atomic with regard to adding subscribers, so between the time the SubmitEvent() API is called, and the time

the local state table is updated, no subscriptions can be added or removed.

2. Meanwhile, a UPnP UCP initializes.
   a. It initializes its HTTP server, passively listening on a TCP port.
   b. If the UCP started up before the UPnP device initialized, it won't see any services become available. When the device finally starts, the UCP will be notified.
   c. Once the UPnP services have been announced the UCP will be able to access one or more of them.
   d. The UCP drives the UPnP API to instantiate a UPnP Service Object.
   e. The UPnP Service Object does the following when it is instantiated:
      i. It obtains the event subscription URL from the description for that service.
      ii. It calls the SSDP API RegisterNotification() specifying SSDP_PROPCHANGE as the event type, the event subscription URL, a callback function pointer (which is a static member function of the class), and a context pointer (which is the "this" pointer of the class). RegisterNotification() does the following:
         1. It makes an LRPC call to the SSDP service. The rest happens on the service side.
         2. If this is the first time it is called for SSDP_PROPCHANGE notifications, RegisterNotification() will call InternetOpen() to get a handle to an internet session. This handle is shared among all local UPnP UCPs.

3. It calls InternetConnect() passing the server name given in the URL it was passed.
4. It calls HttpOpenRequest() passing in the rest of the URL it was passed.
5. The handles returned by these functions are saved with the structure that maintains the subscription.
6. It composes a SUBSCRIBE message, using the data passed in, by calling HttpAddRequestHeaders(). It adds the "NT", "Callback", and "Timeout" headers. The Callback header of the SUBSCRIBE message will be composed on the fly, as an arbitrary URL for notifications to be sent to for this subscription. The server name is the local IP address, and the port is the same one referred to by step 2a above.
7. It calls HttpSendRequest() to send the request to the CD. This is a synchronous function that will return when the request has been responded to by the CD.
8. It calls HttpQueryInfo(..., HTTP_QUERY_CUSTOM, ...) to get the "Subscription-Id" header. The resulting SID will be stored with the subscription structure.
9. It calls HttpQueryInfo(..., HTTP_QUERY_CUSTOM, ...) to get the "Timeout" header. The resulting timeout value will be stored with the subscription structure.
10. A timer is started for re-subscription based on the timeout value returned in the response. When the timer goes off, the re-subscription will be sent.